A Comparison Between Apache Maven and Gradle: Which Is a Better Java Build Automation Tool from the perspective of a Java Developer

YAOWEN MEI, Universe of Waterloo, Canada

Apache Maven and Gradle are the two most commonly used Java build tools. In this paper, an identical java project with 10 unit tests was created for both Apache Maven and Gradle. I then compared the difference between these two tools in the scope of design logic, performance, and user experience. To obtain a quantitative comparison to support my conclusion, I have compared the time consumption of Maven's clean install command vs. Gradle's clean build command with build caching feature enabled. I found Gradle is 60 times faster than Maven in this test, and I also found that the build caching feature provides Gradle the ability to conduct clean builds more efficient for certain users (for example, the developers that need to constantly switch between git branches, or a CI server). Based on my experience, A conclude was made with a SWOT (Strengths, Weaknesses, Opportunities and Threats) analysis for Gradle. The conclusion is that Gradle is a modern build tool and will take over Maven's current market share in the near future

CCS Concepts: • **Social and professional topics** → **History of software**; • **Computer systems organization** → **Architectures**.

Additional Key Words and Phrases: Android Studio, Eclipse, Apache Maven, Gradle, Software Architecture

ACM Reference Format:

Yaowen Mei. 2021. A Comparison Between Apache Maven and Gradle: Which Is a Better Java Build Automation Tool from the perspective of a Java Developer. 1, 1 (August 2021), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Before 2015, the combination of Eclipse + Apache Maven was almost the standard environment for Android app development; while, by the year of 2021, Android Studio + Gradle had taken the place of Eclipse + Maven, and became the most well used Android development environment. Despite of the regional prosperous state of Android Studio + Gradle inside of the Android developers' community, Gradle's global market-share among all the Java Virtual Machine (JVM) developers is surprisingly low comparing to what Maven is holding. According to Snyk's *JVM Ecosystem Report 2020*[1], more than 65 percent of the JVM developers are using Maven as their building tool; while, only 25 percent of them are using Gradle.

On May 16th, 2013, Google released the first early access version of Android Studio in its annually developer conference, I/O 13[4]. Android Studio was designed based on a customised JetBrain's IntelliJ Community Edition software, in the aim of replacing the previously-existing Eclipse Android Development Tools (E-ADT). In the year of 2015, after Google terminated the support for E-ADT, Eclipse foundation released their own E-ADT's replacement, the Eclipse Andmore (but this is not successful on the market). Eight years has been past since the first release of Android Studio. Nowadays, Android Studio is dominating the Android application development market. The combination of Android Studio plus Gradle has pretty much became to the industrial standard regardless of the scale and the scope of an Android

Author's address: Yaowen Mei, ywmei@uoguelph.ca, Universe of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1.

Manuscript submitted to ACM

development project. Interestingly, among all JVM developers, the market-share of Maven has always been 3 times as high as what Gradle holds since the year of 2012. These contradictory will naturally lead to the following two questions:

(1) What are the fundamental difference between Apache Maven and Gradle in terms of design, features and performance? Is these difference lead Google to choose Gradle as the default building tool for Android studio;
(2) What software development architecture patterns Maven and Gradle are following? Which one is better?

In order to answer these two questions above, the "Advanced Software Development Tools", Gradle and Apache Maven, have been used and the different features between them have been explored. The differences between Gradle and Apache Maven are described in the following sections in this report in terms of 1) design logic, 2) efficiency, and 3) user experience. In the conclusion section, a SWOT (Strengths, Weaknesses, Opportunities and Threats) analysis for Gradle is provide.

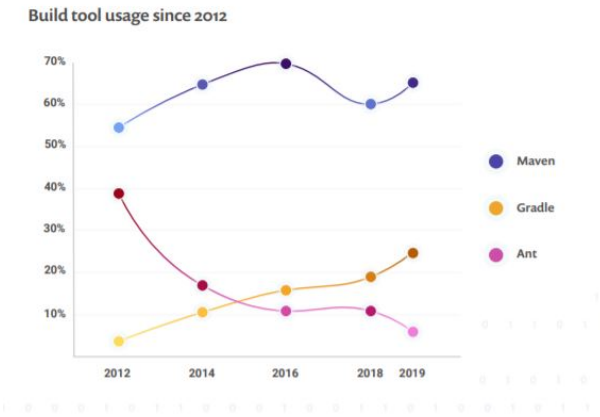


Fig. 1. The historical build tools' market-share data among JVM users. Directly copied from Snyk's website [https://snyk.io/], (https://snyk.io/wp-content/uploads/jvm_2020.pdf).

2 DIFFERENCES IN DESIGN LOGIC

Both Apache Maven and Gradle are the first-class software development tools. However, there are many fundamental differences between these two tools. First of all, the design logic between them is different. Compared to Gradle, Maven is a more rigid tool in terms of project folder structure, build task life cycles and build script languages. In order to explore these differences, a basic "Hello World" java project with 10 unit tests was created with IntelliJ, and then, this project was manually converted to Maven project and Gradle project separately. The 10 unit tests are setted in the way that test0.java is 0 millisecond delay with a assert statement that is always true, test1 is 1000 millisecond delay with a assert statement that is always true, test2 is 2000 millisecond delay with a assert statement that is always true and so on and so forth, so that I can use these two projects to explore the efficiency difference between Maven and Gradle in the next section. These two projects can be found in this Github link:

2.1 Different Folder Structures

During the time when I was trying to manually convert the plain "Hello World" Java project into the Maven and Gradle project, the first thing I noticed is that Maven has a very strict standard folder structure, and any maven project must

follow this layout. In contrast, Gradle is naturally supporting Maven's folder structure[2], but Gradle also naturally supports customized folder structure. one can easily modify the folder structure with just a few lines of code in Gradle's build scripts[9]. As blue color text indicated in figure 2, Gradle project does not necessarily to have a folder structure. With the sourceSets command, one can set or add the main folder and the test folder to anywhere on the hard disk.

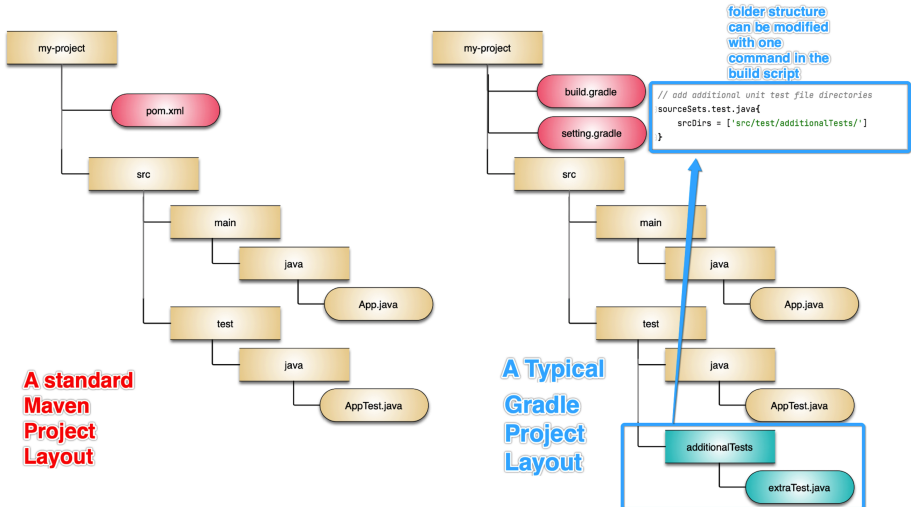


Fig. 2. The folder structure of the basic "Hello World" Maven and Gradle project used for this article. The red pom.xml file on the left panel uniquely define a Maven project, while the two red files, build.gradle and setting.gradle, uniquely define a Gradle project.

2.2 Different Build Scrip Languages

The Maven's build file is written in a rigid Extensible Markup Language (XML), and the file is named as POM.xml. Due to the fact that this file is very rigidly defined, the learning curve for maven is flat, even beginners can easily read an existing Maven project as long as he/she does not need to do too much customization on the project. However, as the old saying says: "one's greatest disadvantage is his advantage", a great number of Java developers might even think this POM.xml file is tediously long and is not flexible. For example, If a developer is not trained on how to write Maven plugins, then this developer's building ability will be limited to whatever plugins are available in the Maven ecosystem. One of the good things about this is that due to the longer tenure of Maven, there are a great variety of plugins in the market already, but still, the rigidity brings a barrier for customization and automation of the maven building process. On the other hand, Gradle supports the Domain Specific Language (DSL) in its build files[7]. Gradle's build scripts support two kinds of JVM languages, Groovy and Kotlin. Groovy DSL and Kotlin DSL provide good inter-compatibility and interoperability to Gradle build scripts and make it more flexible. By utilizing these DSLs, if one wants to create custom tasks, he/she can do it right inside of the build.gradle file instead of creating a plugin for Gradle.

The most important portion of the "Hello Word" Maven and Gradle build files are shown in Fig.3, as we can see from this direct comparison between Maven's POM.xml file with Gradle's build.gradle file, Maven on average takes 3 times more code than Gradle to accomplish the same task. This tedious makes Maven build file hard to read for big

projects. Please note that the last row shows the gradle can elegantly change the project folder structure by just one line of code, and this is also demonstrated in Fig.2 within the blue color boxes.

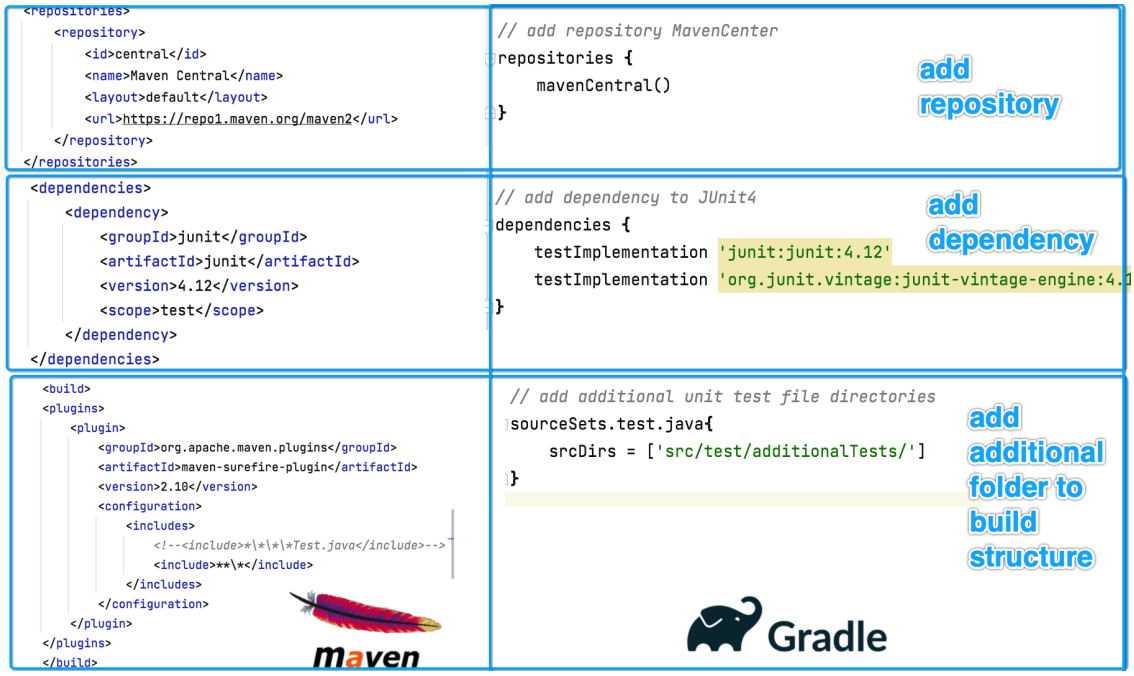


Fig. 3. The snapshot of critical build file sections from the basic "Hello World" Maven and Gradle project used for this article. The right panel shows 3 functionalities (add repository, add dependency, and modify project folder structure) implemented by Gradle's DSL language. The left panel is the same functionalities implemented by Maven's XML file.

3 DIFFERENT PERFORMANCE

In this section, I will discuss the performance difference between Maven and Gradle in terms of life cycles design and incremental building strategy. At the end of this section, utilizing the project files as described in the previous section, a quantitative analysis of each of these two build tool's building efficiency is presented.

3.1 Life Cycles

In order to compare the performance difference between Maven and Gradle, the life cycles in each of the Maven build and Gradle build must be understood. In Maven, there are many different phases responsible for different tasks, and these phases can be classified into 3 distinct life cycles during a typical Maven build[10]. These life cycles are

(1) the clean life cycle that cleans the old files from the output directory;
(2) the default life cycle during which the project is build from validate phase all the way down to the deploy phase[3];
(3) the site life cycle during which a Maven project's site document is generated.

Each of these phases consist of a sequence of goals and each goal represents a specific task. A maven plugin is also basically a sequence of goals that can provide Maven specific functionalities. Different plugins require different

dependencies (Jar files) that need to be added to the classpath. Maven therefore provided six dependency scopes, namely to be compile, provided, runtime, test, system, and import so that user can specify during which scope the dependency should be added to the classpath.

In contrast, the building lifecycle of a Gradle project can be classified into 3 phases:

(1) the initialization phase, during which Gradle first uses the gradle.properties file to conduct the environment configuration, and then Gradle uses the setting.gradle file to find the root of the project as well as all the related subprojects (a Project instance is created for each of the projects/subprojects);
(2) the configuration phase, during which Gradle evaluates all the build scripts from build.gradle files, then builds the object model (adding customized functionalities to each of the Project instance based on the content of each of the build.gradle files), then, generates a directed acyclic graph (DAG) to represent the execution sequence of all the tasks which are involved in this build;
(3) the execution phase, during which Gradle executes all the necessary tasks along the DAG according to the argument passed to the gradle command in the command line/user input.

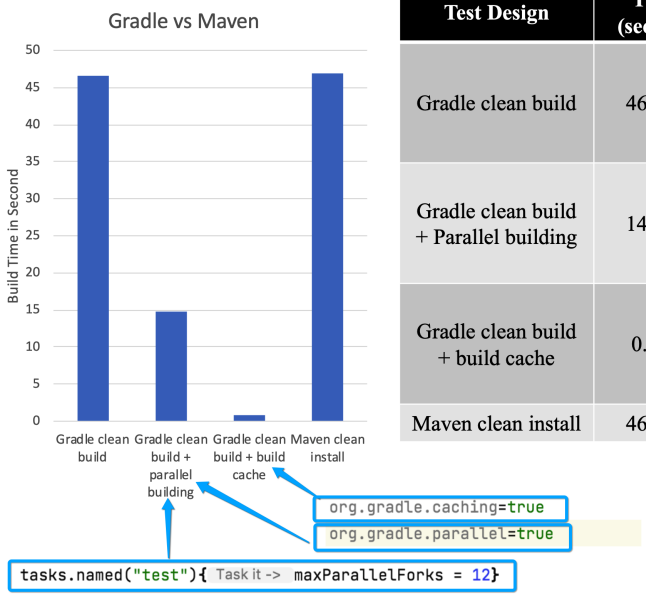
As a summary, during the initialization phase, Gradle try to find out which folders/files it need to process for the build, then in the configuration phase, Gradle actually scans all the folders/files that need to be processed and then generate the execution DAG of all related tasks, finally, in the execution phase, with the help of the DAG, Gradle will only execute the minimum number of tasks that adequate to accomplish this specific build. Due to the fact Gradle is organizing the build activity based on the DAG of tasks[8], while Maven is organizing the build activity based on all the the goals attached to different phases in the pom.xml file, the life cycle design of Gradle is more advanced than Maven and therefore, Gradle is more efficient.

3.2 Incremental Building

For every task, Gradle naturally allows users to specify certain properties designated as inputs and outputs (Maven need to use something like a Gradle plugin to achieve this). Gradle can utilize these inputs to skip certain tasks, and this is the so-called Gradle's "incremental build" property. When a user designated a field as input and another field as output, Gradle will make a hash and save the value of them. Later on, when the user wants to run the task again, Gradle finds out that the input has not changed, and the output still exists, then Gradle knows that this task can be safely skipped[5]. By doing so, Gradle is more efficient than Maven in terms of running time. Based on the concept of incremental building, there are 3 unique features that Gradle is providing to users, and make Gradle faster than Maven:

(1) **Build caching**: Gradle can skip the execution of some tasks if it has been done before, even if these tasks are done on other computers or the CI server (distributed caching).
(2) **Parallel building**: Gradle can execute sub projects in parallel if Gradle find that they are independent with each other.
(3) **File watching**: Gradle can watch certain files on the disk and when changes are detected, Gradle will automatically rebuild.

3.2.1 Build Caching. One of the features that makes Gradle faster than Maven is the build caching. This term was introduced to the Java building world by Gradle around the year of 2017. In Gradle, this feature comes with default, but Maven users can also use this feature, but they need to use the Gradle Enterprise Plugin and pay the subscription fee to Gradle company for this feature. Both Maven and Gradle have the dependency cache (the .m2 folder for Maven and



Test Design	Time (second)	Link to the report	Time Saving From
Gradle clean build	46.553	https://enterprise-training.gradle.com/s/kaqp3akwvqy6i/timeline	NA
Gradle clean build + Parallel building	14.792	https://enterprise-training.gradle.com/s/gz5glkdsra5hu/timeline	running all the 10 tests in parallow
Gradle clean build + build cache	0.790	https://enterprise-training.gradle.com/s/tkkgusgoushk6/timeline	directly using all the build cache from the previous build
Maven clean install	46.864	NA	NA

Fig. 4. The build time comparison of the basic "Hello World" Maven and Gradle project with 10 unit tests.

the .gradle folder for Gradle), which saves the binaries that represent all the required dependency repositories on the local machine for a build. However, build caching is different from a conventional dependency cache in the following aspects: 1) build cache records and saves a series of build actions, 2) build cache can speed up the build for a single source repository. A typical example is Alice and Bob are working on the same project but on different branches, they work and build on their own computer. When Alice is reviewing and testing Bob's code, Alice will need to switch to Bob's branch, download Bob's code, and build the project. The problem happens when Alice finishes reviewing Bob's code and switches back to her own branch. If Alice is using Maven, or does not have build cache enabled in Gradle, Alice will be forced to build her own branch again. However, if Alice enabled the build cache feature in Gradle, then Gradle will keep a copy of the build tasks before her switch to Bob's branch, and so when Alice switches back to her own branch, Gradle just provides her the build result directly from the cache.

3.2.2 Parallel building. Gradle is naturally supporting structuring builds (parallel building multiple projects) as well as supporting parallel executing unit tests. In software architecture level, there are two types of structuring builds, one type is called the multi-project builds, where there are many sub projects inside of one root project, the sub projects are connecting to the root project with low coupling and high cohesion. The other type of structuring build is called the composite builds where a group of sub projects are released together, but each one of them is more like a separate component, and may rely on the output of the other component. Gradle's parallel building strategy can speed up the first type, but there is not much a building tool can do with the second type.

The requirement for this article is to use a advanced software tool and summarize the user experience. In order to satisfying the requirement, and to have a quantitative evaluation of the efficiency of Maven and Gradle, I have designed

a comparison test between Maven and Gradle using the Maven project and the Gradle project described in the previous Section, Fig2 and Fig3. In this test, Maven's building speed is compared with Gradle under the condition of:

(1) build cache is disabled;
(2) only build cache is enabled;
(3) only parallel building is enabled.

As the result shown in Fig.4, without activating the incremental building feature, Gradle clean build and Maven clean install almost take the same time to finish the job (46.553 second for Gradle vs. 46.864 for Maven). With the activation of the parallels building, with 12 parallel forks working together, the Gradle's clean build time reduced to 14.792 seconds. This result is reasonable as all of our 10 test files, as described in the introduction section, are all independent with each other. After assigning 12 JVM to the build, each test will have its own JVM, so the total test running time will equal to the longest running test, which is the test9.java with 9 seconds sleep. Due to the lenght requirement of this article, I didn't explore the parallel test running feature in Maven, but I would expect the result will be similar with this Grdle result[6] Another most significant result is with the build cache enabled, Gradle's clean build speed is almost the same as regular build with local cache (less than 1 second), and Gradle's clean build speed is 60 times faster than Maven's clean install. This test result indicates that Gradle's efficiency is better than Maven in case one need to conduct clean build frequently. For example, if a small company has 10 developers, and each of them need to conduct 100 times clean build per week. If Gradle can save 30 seconds per build, then Gradle will save the company 18 days of work per year. On the other hand, less waiting time can promote more local build, and increase the over code quality of the company. Therefore, Gradle as a faster build tool can bring a lot of benefit to the end user, either as developers or as company owners.

4 USER EXPERIENCE

Both Maven and Gradle can provide a build summary document at the end of each build. Maven use the command "mvn site" and Gradle use the command "gradle build –scan". Maven is presenting this summary information as a html web-page file inside of the target folder, while, Gralde is both saving a quick summary html web-page locally as well as is generating a more detailed build report on Gradle.com's company server. As shown in the Fig.5, the Maven site is more like a Java Doc, it shows the user some basic information like, all the plugins, all the dependencies, and repositories related to the build. The red box in the Fig.5 shows that Maven can tell user the version of each of the plugins used in this project. In a Gradle scan report, almost everything is clickable. It has a modern UI design, and it is more focused on the building time. As shown in the blue box in Fig.5, Gradle tells you the order of each of the tasks involved in this build, tells you start time of each task, the duration of each task, and also tells you if this task is executed as fresh during the build, or is get from the local cache of last build (UP-TO-DATE), or from the build cache (FROM CACHE). Overall, Gradle's build scan is more informative, and can really help developers a lot when they do trouble shooting for a problem either on local build or CI build,

5 GRADLE AND ANDROID

Every Android project is a multi-project build, the user of android studio gets a top level root project named as the application's name (a root level setting.gradle and a root level build.gradle file associated with the root level project), and then a sub project called app (this is where we actually write the code for our android app). As we seen in Section 3, Gradle can really speed up the multi-project build. I believe this is part of the reason why google want to choose Gradle

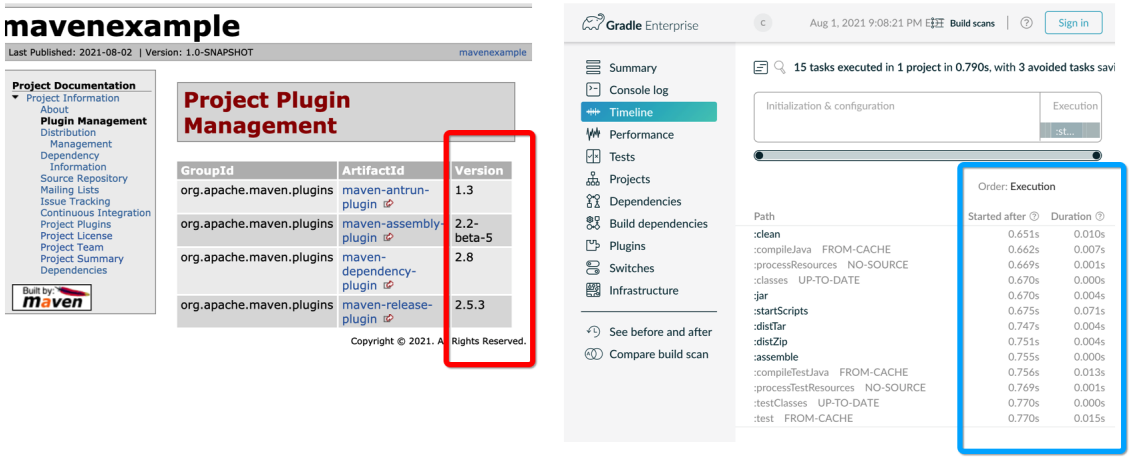


Fig. 5. The layout of a Gradle site and a Gradle scan report. The left panel is the maven site, and the right panel is the Gradle scan report

as Android Studio's default build tool. At the end of this article, I want to emphasized on the difference betwen the project level build.gradle file and the module level build.gradle file for any android application.

5.1 The root level build.gradle file (project level)

The first line of the root level build.gradle file is the "buildscript" code block instead of a conventional "plugin" code block, this is due to the fact that this Android Gradle plugin is not registered in the Gradle Plugin repository, in this way, Google can have the control of this plugin and update it together with each Android studio in the "lock-step" manner . The root level build.project file is modifying/adding plugins to Gradle itself. This file is not for modifying the user's android app's dependencies.

5.2 The sub-project level build.gradle file (module level)

While, if we go inside of the app folder, the build.gradle file inside of the app folder starts with the plugins code block, and this is the module level build file for this particular sub project. As shown in the picture, the second code block starts with "android" is purely from the com.andorid.application plugin, this is to say, without this plugin, Gradle will not be able to understand anything inside of this android code block.

6 SUMMARY

In summary, Gradle is a more flexible software build tool than Maven in terms of folder structure and build script language. The design of life cycles is different between Maven and Gradle and this leads to the difference in building logics. Maven's building customization has to be achieved via the Maven plugins, while each of the plugins need to be attached to a certain Maven phases; while, Gradle's plugin can bring additional functionality to Gradle, and these additional functionalities are called tasks. Customization DSL can be attached to the tasks and will be invoked together with the tasks. However, the sequence of which task will be executed first is dependent on the DAG generated from the Gradle's configuration phase. Gradle is more efficient than Maven in terms of building speed. In the test I designed and

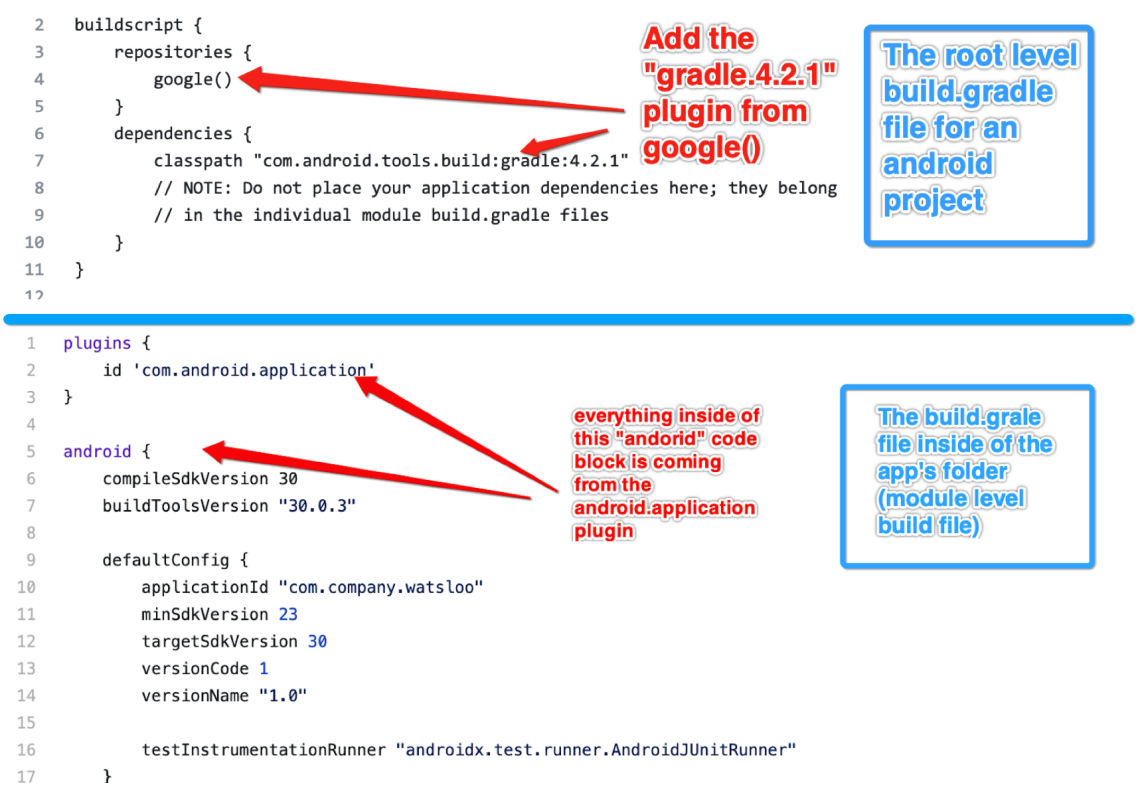


Fig. 6. The project level and the module level build.gradle file for an Android project. The top panel is the project level one. The purpose of this file is to modify Gradle itself, and this modification will be inherited by all the sub projects. The bottom panel is the module level build.gradle file. The purpose of this file is to modify the Android app.

conducted, I found that Gradle is 60 times faster than Maven to conduct a clean install task. The reason for this is that Gradle allows a developer to use cache from the build which was done several days ago, either on his local machine, or the CI server, or even on his colleague's machine. Maven actually can also use this build cache feature if the maven developer pays the money and uses the Maven's Gradle Enterprise plugin. Last but not the least, Gradle provide a very informative build report including almost all the information about the build, and one can easily conduct trouble shooting with this build scan report. The SWOT analysis of Gradle is presented below in Fig.7. In conclusion, based on the design logic, efficiency, and user experience, I think Gradle should be a better build tool for large projects which includes a lot of sub-projects inside. While Maven is better for small scale projects due to Maven's flat learning curve. I believe that the emergence of Gradle is not just a simple "reinventing the wheel" from Maven. Based on the nature of Gradle, Gradle is truly a revolution from Maven, and I think Gradle is going to replace Maven in the very near future.

REFERENCES

[1] Brian Vermeer February 5. 2021. JVM Ecosystem Report 2020. https://snyk.io/blog/jvm-ecosystem-report-2020/
[2] Tim Berglund and Matthew McCullough. 2011. *Building and testing with Gradle.* " O'Reilly Media, Inc.".
[3] Casimir De´sarmeaux, Andrea Pecatikov, and Shane McIntosh. 2016. The Dispersion of Build Maintenance Activity across Maven Lifecycle Phases. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR).* 492–495.

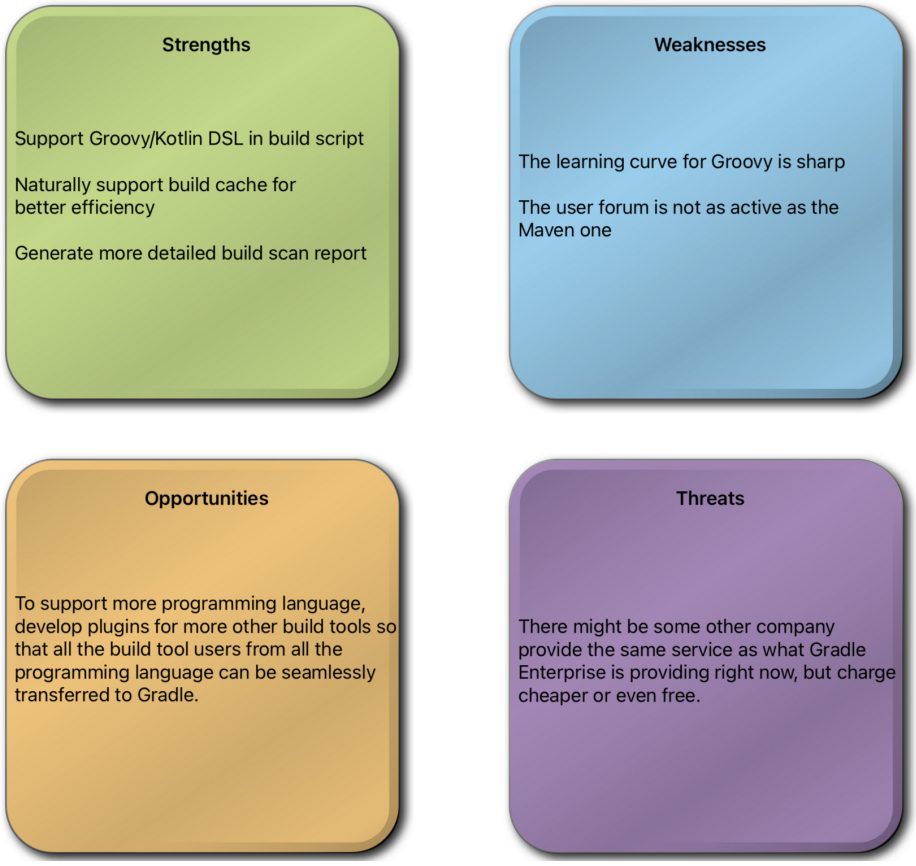


Fig. 7. The SWOT analysis for Gradle

[4] Xavier Ducrohet, Tor Norbye, and Katherine Chou. 2013. Android Studio: An IDE built for Android. https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html
[5] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*. 94–106.
[6] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 249–259.
[7] MA Krishna Priya and Justus Selwyn. [n.d.]. CODE PRESENCE USING CODE SENSE. ([n. d.]).
[8] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A model for detecting faults in build specifications. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[9] Balaji Varanasi. 2015. Dependency Management. In *Introducing Gradle*. Springer, 67–85.
[10] Balaji Varanasi. 2019. Maven Lifecycle. In *Introducing Maven*. Springer, 53–68.