

Playing with Gradle

An Android vision on Gradle
through an application with flavors
with an industrialized delivery goal



Code is on GitHub

<https://github.com/MathiasSeguy-Android2EE/MultiplicationBasile>



*By Mathias Seguy
also known as Android2ee*



Table of contents

Playing with Gradle	1
Chapter 1: Introduction.....	5
1 Gradle in the project	5
1.1 Gradle's configuration files	5
1.2 Project's configuration files.....	9
1.3 Gradle with the command line.....	12
Chapter 2: Android tasks.....	14
2 Android tasks.....	14
2.1 By families	16
2.2 Android tasks family.....	17
2.3 Build tasks family.....	21
2.4 Install tasks family	24
2.5 Verification task family.....	25
2.6 Others details	25
3 Simpler with AndroidStudio	26
4 Conclusion	28
Chapter3: Flavors, BuildTypes and Variants.....	29
5 BuildTypes	29
6 Flavors	30
6.1 Basics	30
6.2 Dimensions	35
7 Step by step example	37
7.1 Gradle file	37
7.2 Setting your folders structure	39
7.3 Customizing Resources.....	39
7.4 Customizing Java code.....	49
Chapter 4: Flavors And Manifest.....	52
8 In practice.....	52
8.1 The getInstance question	55
Chapter 5: Understanding the life cycle.....	57
9 The problem	57
10 Comprehension	58
11 Solution.....	58

12	Conclusion	58
Chapter 6: Setting code coverage on Android with Jacoco.....		59
13	Enabling Jacoco on your project.....	59
13.1	Enable code coverage for instrumented tests	59
13.2	Enable tests coverage for UnitTests.....	60
13.3	What happened.....	60
14	Defining and creating your report.....	62
15	References.....	64
Chapter 7: Build Organization		65
16	Builds Organization.....	66
16.1	General notions on *.properties files.....	66
16.2	Defining your own properties files.....	67
16.3	Extracting password from gradle.properties file (Default)	67
16.4	Extracting password from my gradle_others.properties	69
16.5	Splitting the build.gradle file into several files to gain in readability.....	72
16.6	Define variables for Java code and Manifest	73
16.7	Custom my gradle memory to run fast	75
17	Application's builds	76
17.1	Mock and Production flavors for tests environments.....	76
17.2	Deliver the project for weekly delivery to the team and stakeholders	78
Chapter IX: Getting real.....		80
18	Organize your Gradle files as your code.....	80
19	Extract your build variables and constants	81
20	Load your external properties files	82
21	Extract your hooks.....	84
22	Building, testing and Analyzing Script	86
22.1	BuildAndCheckProject task.....	86
22.2	fullBuild task.....	87
22.3	runReporters task.....	87
22.4	runReporters:FindBug task.....	89
22.5	runReporters:Jacoco task	91
22.6	runReporters: JavaDoc task.....	95
22.7	runReporters: PMD task.....	96
23	The release scripts.....	98
23.1	One entry page for your report.....	101
24	The upload part	104

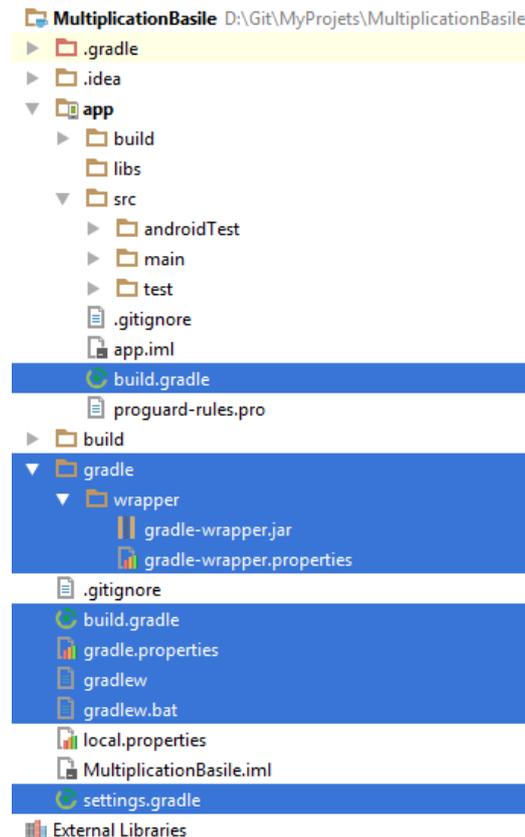
24.1	Upload: Maven basics principles	104
24.2	Upload : In a Nexus Repository	107
25	Taking care of libraries	111
26	Conclusion	112

Chapter 1: Introduction

Let's have a look at the build system use for developing Android application.

1 Gradle in the project

In our Android project Gradle is set in the following way:



Yep, gradle is everywhere, scaring isn't it?

By the way there are 2 different types of files:

- Those that describe the project
- Those that describe the gradle's configuration to use for this project.

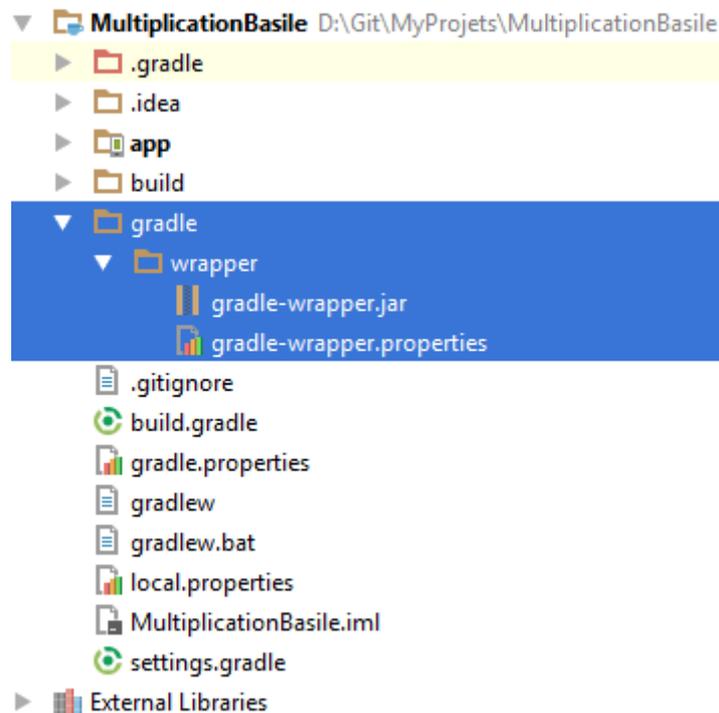
Let's beginning by the gradle's configuration.

1.1 Gradle's configuration files

1.1.1 What Gradle's version should I use

The first parameter you have to define is which version of Gradle you want to use. In a way, it's pretty natural and well thought, don't you think?

The files concern are those ones:



Which is the gradle folder of the project but more specifically is where you define the wrapper to retrieve the version of gradle you want to use.

To define the gradle version, we just define where is the version of Gradle to use (and download if it's missing) in the gradle-wrapper.properties:

#Mon Dec 28 10:00:20 PST 2015

```
distributionBase=GRADLE_USER_HOME  
distributionPath=wrapper/dists  
zipStoreBase=GRADLE_USER_HOME  
zipStorePath=wrapper/dists  
distributionUrl=https://services.gradle.org/distributions/gradle-3.3-all.zip
```

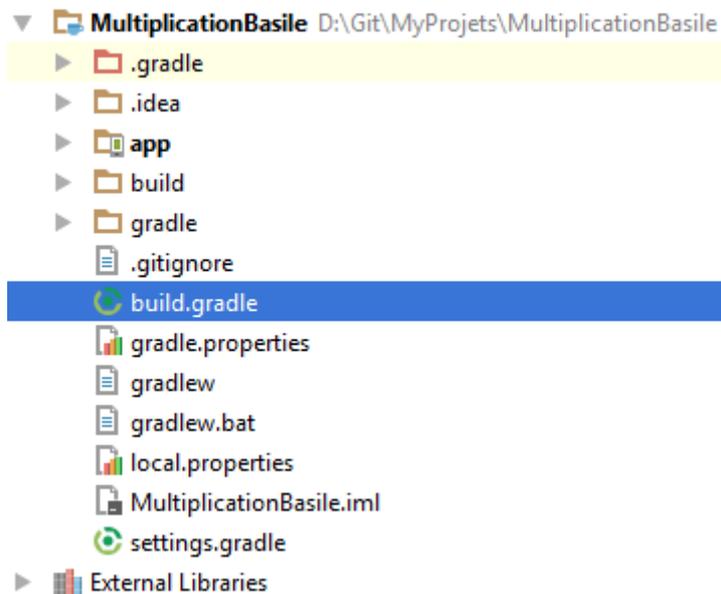
In this file you just have to change the url to the version you want to use. You can have a look here to see what is the last version number: <http://services.gradle.org/distributions>

A good rule for that is to have the latest version of gradle because the gradle team works hard to optimize your builds from version to version.

1.1.2 How can I build this type of project?

The second element to configure is what type of project do I build and more specifically how can I build them? Because each type have specific tasks and way to do the build. Those specific tasks are delivered through gradle's plugins. So we need to explain to gradle where it can find the plugins it needs.

We do that in our root build.gradle file. We call it like that because it is at the root of your project and defines properties for the project and all its subprojects.



This file is the following

// Top-level build file where you can add configuration options common to all sub-projects/modules.

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.3'

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

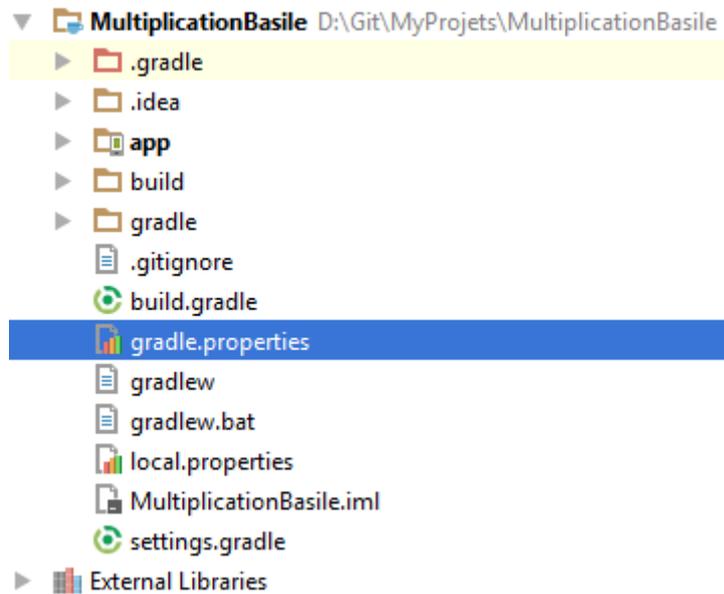
You have 3 different parts:

- The buildscript where you define where the repositories are and what is the names of the gradle's plugins to use for this project.
- allProjects define for all the project and its sub-projects where to download the libraries it needs
- task is just a new task added to your gradle task that just clean the build directory

So here, we just define that we want to use com.android.tools.build:gradle:2.2.3 as plugin for gradle to build our project. It means the tasks of this plugin will be available for building your project.

1.1.3 How can I tune gradle (memory and more)?

In the file `gradle.properties` you can define a lot of properties for your project (and its sub-projects).



We often use this file to define the memory dedicated to the gradle daemon.

```
# Project-wide Gradle settings.
```

```
# IDE (e.g. Android Studio) users:  
# Gradle settings configured through the IDE *will override*  
# any settings specified in this file.
```

```
# For more details on how to configure your build environment visit  
# http://www.gradle.org/docs/current/userguide/build\_environment.html
```

```
# Specifies the JVM arguments used for the daemon process.  
# The setting is particularly useful for tweaking memory settings.  
#org.gradle.jvmargs=-Xmx1536m
```

```
# When configured, Gradle will run in incubating parallel mode.  
# This option should only be used with decoupled projects. More details, visit  
# http://www.gradle.org/docs/current/userguide/multi\_project\_builds.html#sec:decoupled\_projects  
# org.gradle.parallel=true
```

```
#Enable daemon  
org.gradle.daemon=true
```

```
# Specifies the JVM arguments used for the daemon process.  
# The setting is particularly useful for tweaking memory settings.  
# Try and findout the best heap size for your project build.  
org.gradle.jvmargs=-Xmx2048m -XX:MaxPermSize=512m -XX:+HeapDumpOnOutOfMemoryError -  
Dfile.encoding=UTF-8
```

```
# When configured, Gradle will run in incubating parallel mode.  
# This option should only be used with decoupled projects. More details, visit  
# http://www.gradle.org/docs/current/userguide/multi\_project\_builds.html#sec:decoupled\_projects  
# Modularise your project and enable parallel build  
org.gradle.parallel=true
```

```
# Enable configure on demand.  
# avoid building part of the project when it's not necessary  
org.gradle.configureondemand=true
```

In this example we have define that:

- the configuration on demand is enable
- we run the build in parallel
- we want 2048 M of JVM for the daemon
- we want the daemon to be activated

So we tune our gradle process depending on our computer to optimize it.

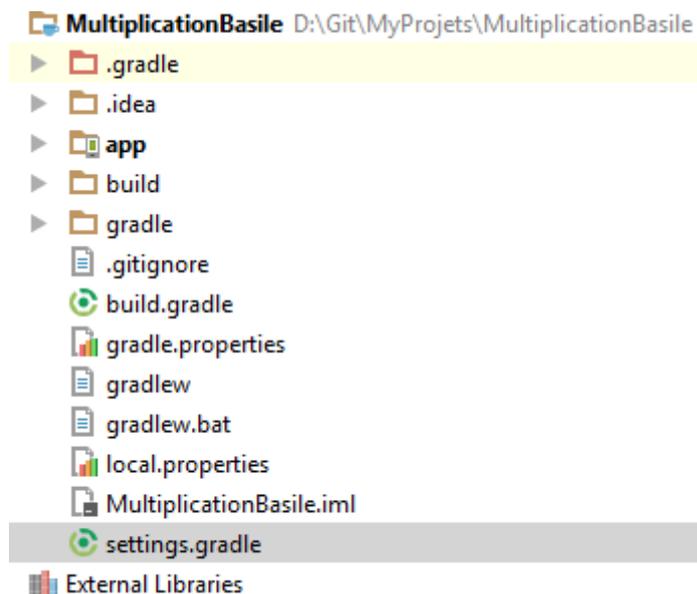
We will see that file is also use to define others types of variables that will be accessible by the subprojects.

1.2 Project's configuration files

So we have explained to gradle how to work, let's explain to it what to do.

1.2.1 How many modules belong to your project?

We first describes what contain our project. This is done in the setting.gradle file:

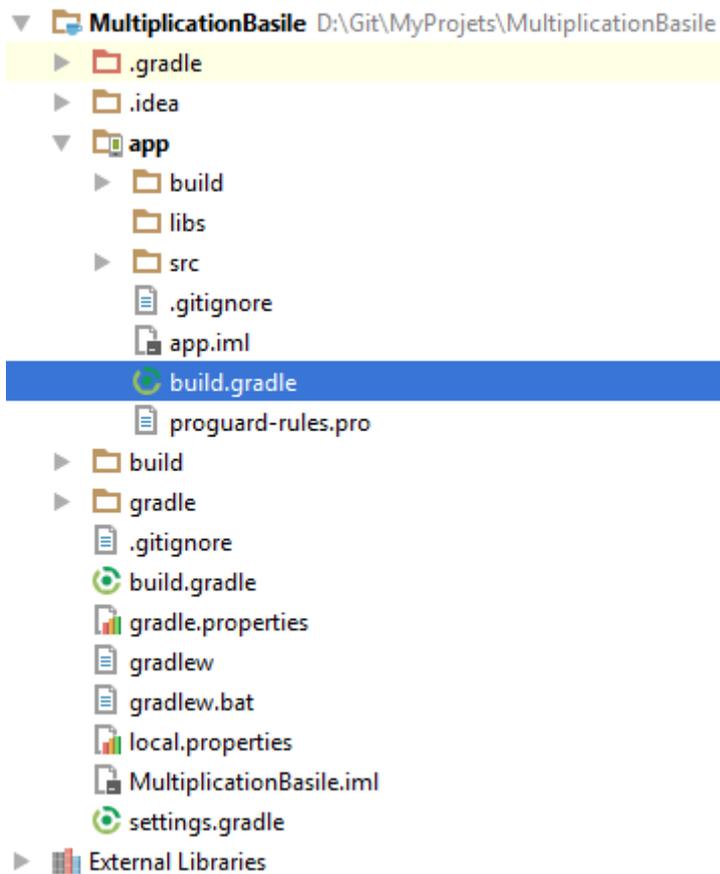


And it only contains the list of your sub-projects:

```
include ':app'
```

1.2.2 What is my module(s) description?

Now we are ready to define our Android project. This description is done in the file build.gradle at the root level of your module.



And it contains:

apply **plugin: 'com.android.application'**

```

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.android2ee.basile.multiplication"
        minSdkVersion 10
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.1.0'
}

```

```
testCompile 'junit:junit:4.12'  
}
```

This file contains 3 main parts:

- What is the plugin to use to build the module: we want android plugin
- What is the android description of the module: the android block
- What are the libraries I need to build it: the dependencies block

The android block defines essential elements of your Android application:

- compileSDK: what version of the SDK do I use for compilation, higher is better
- minSDK: What is the minimal SDK version am I compatible with. A good rule is to follow GoogleServices minSDK
- targetSDK: What is my running environment of predilection (If I run in a higher version of it, the system will emulate my target version has execution environment).
- versionCode : Auto-increment integer that follow the delivered version
- versionName: Human readable version
- applicationId: We used to have the root package of the application as applicationId and to define it in the manifest.xml. Now you can keep your root package in the manifest (it will be use by the device when running your application for the classpath) and have a different applicationId in your build.gradle (it will be used by PlayStore and Device to identify your application). That means you can refactor your packages and keep your applicationId.
- testInstrumentationRunner defined which instrumentation runner you want to use when running your test. It's related to the AndroidTestingSupportLibrary Sdk and give you the ability to use a Junit version 4.

The build type block gives you the ability to specify for each build type specific elements (like versionNameSuffix, applicationIDSuffix, minify, shrink...).

By default you have 2 build types : Release or Debug. And you should stick on them because they are the way to build your application. If you want to customize your application according to a context (client, processor, brand, free/paid,...) you should consider flavors.

The dependencies blocks is here to describes what are the libraries you depends on. You can specify if it's for compilation, test, for which flavor... and so on. Let's have a look:

compile fileTree(dir: 'libs', include: ['.jar'])* explains that all the files contained in the folder libs that ends with .jar should be included for compilation of the apk,

```
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {  
    exclude group: 'com.android.support', module: 'support-annotations'  
})
```

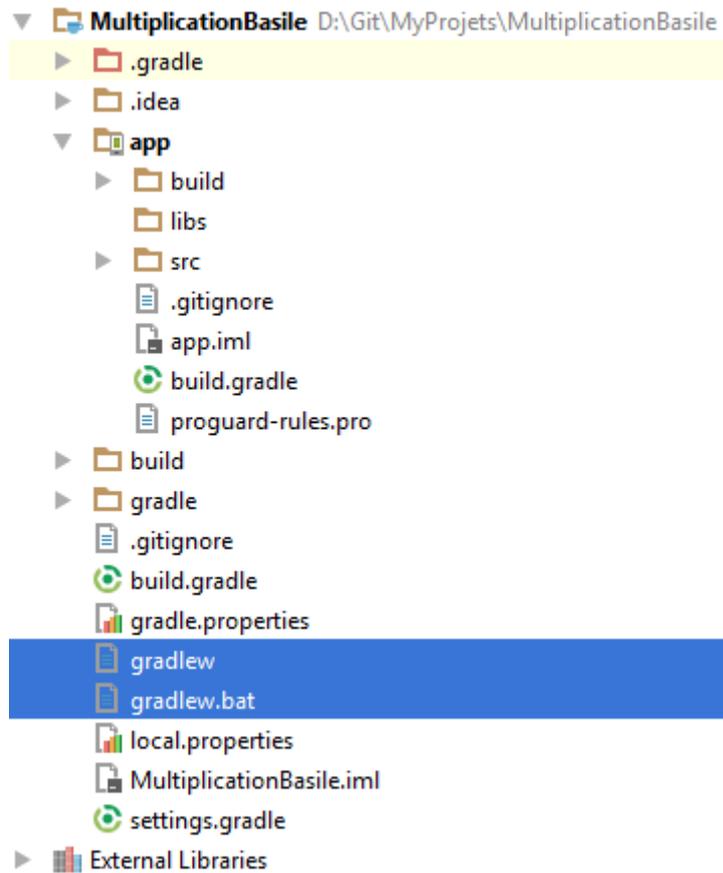
explains that when running for instrumentationTest we will need espresso but we will exclude the support-annotations modules from it (transitive dependencies)

compile 'com.android.support:appcompat-v7:25.1.0' explains that we use the artifact with groupId=com.android.support, artifactId=appcompat-v7 with version 25.1.0. Which is the normal coordinate of any project in maven repositories.

testCompile 'junit:junit:4.12' just explains that for unit tests we need Junit 4.12 expressed in maven's project coordinates.

1.3 Gradle with the command line

Here is the last files you find in your project gradlew and gradlew.bat which are the gradle worker executable.



This is the command line tool (one for linux/mac, the other for windows). That means you can run in command line gradle command directly from Android Studio like that:

```
Terminal
+ $PATH$
X math@LAPTOP-SUBSQ2NN MINGW64 /d/Git/MyProjets/MultiplicationBasile
$ gradlew version
bash: gradlew: command not found

math@LAPTOP-SUBSQ2NN MINGW64 /d/Git/MyProjets/MultiplicationBasile
$ ./gradlew -v

-----
Gradle 3.3
-----

Build time:   2017-01-03 15:31:04 UTC
Revision:     075893a3d0798c0c1f322899b41ceca82e4e134b

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_101 (Oracle Corporation 25.101-b13)
OS:           Windows 10 10.0 amd64

math@LAPTOP-SUBSQ2NN MINGW64 /d/Git/MyProjets/MultiplicationBasile
$ █
```

As you see, use `./gradlew` (relative path from the console point of view) because gradlew is not a global command for your computer.

Chapter 2: Android tasks

Ok, so we had a small introduction with Gradle, now let's dig into the subject a little bit more, we gonna talk about Android tasks to have a better vision on what is it. If you are trying to find tips for your daily code, just skip this chapter.

2 Android tasks

What is a task? a task is just a Gradle function, inherits from others, have parameter that can be others methods and is designed to make a stuff. Blurry definition... by the way the one of the gradle documentation is perhaps for you more clear :

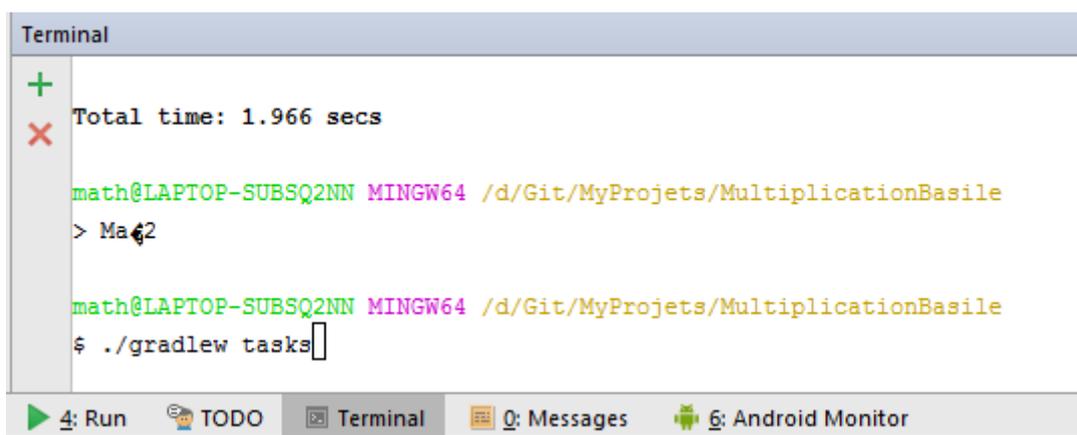
Everything in Gradle sits on top of two basic concepts: [projects](#) and [tasks](#).

Every Gradle build is made up of one or more [projects](#). What a project represents depends on what it is that you are doing with Gradle. For example, a project might represent a library JAR or a web application. It might represent a distribution ZIP assembled from the JARs produced by other projects. A project does not necessarily represent a thing to be built. It might represent a thing to be done, such as deploying your application to staging or production environments. Don't worry if this seems a little vague for now. Gradle's build-by-convention support adds a more concrete definition for what a project is.

Each project is made up of one or more [tasks](#). A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

https://docs.gradle.org/current/userguide/tutorial_using_tasks.html

So if your run `./gradlew tasks` in your terminal you will see the list of tasks available for your project :



```
Terminal
+
X Total time: 1.966 secs
math@LAPTOP-SUBSQ2NN MINGW64 /d/Git/MyProjets/MultiplicationBasile
> Ma42
math@LAPTOP-SUBSQ2NN MINGW64 /d/Git/MyProjets/MultiplicationBasile
$ ./gradlew tasks
```

All tasks runnable from root project

Android tasks

androidDependencies - Displays the Android dependencies of the project.
signingReport - Displays the signing info for each variant.
sourceSets - Prints out all the source sets defined in this project.

Build tasks

assemble - Assembles all variants of all applications and secondary packages.
assembleAndroidTest - Assembles all the Test applications.
assembleDebug - Assembles all Debug builds.
assembleRelease - Assembles all Release builds.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
clean - Deletes the build directory.
compileDebugAndroidTestSources
compileDebugSources
compileDebugUnitTestSources
compileReleaseSources
compileReleaseUnitTestSources
mockableAndroidJar - Creates a version of android.jar that's suitable for unit tests.

Build Setup tasks

init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks

buildEnvironment - Displays all buildscript dependencies declared in root project 'MultiplicationBasile'.
components - Displays the components produced by root project 'MultiplicationBasile'. [incubating]
dependencies - Displays all dependencies declared in root project 'MultiplicationBasile'.
dependencyInsight - Displays the insight into a specific dependency in root project 'MultiplicationBasile'.
dependentComponents - Displays the dependent components of components in root project 'MultiplicationBasile'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'MultiplicationBasile'. [incubating]
projects - Displays the sub-projects of root project 'MultiplicationBasile'.
properties - Displays the properties of root project 'MultiplicationBasile'.
tasks - Displays the tasks runnable from root project 'MultiplicationBasile' (some of the displayed tasks may belong to subprojects).

Install tasks

installDebug - Installs the Debug build.
installDebugAndroidTest - Installs the android (on device) tests for the Debug build.
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build.
uninstallDebugAndroidTest - Uninstalls the android (on device) tests for the Debug build.
uninstallRelease - Uninstalls the Release build.

Verification tasks

check - Runs all checks.
connectedAndroidTest - Installs and runs instrumentation tests for all flavors on connected devices.
connectedCheck - Runs all device checks on currently connected devices.
connectedDebugAndroidTest - Installs and runs the tests for debug on connected devices.
deviceAndroidTest - Installs and runs instrumentation tests using all Device Providers.
deviceCheck - Runs all device checks using Device Providers and Test Servers.
lint - Runs lint on all variants.
lintDebug - Runs lint on the Debug build.
lintRelease - Runs lint on the Release build.
test - Run unit tests for all variants.
testDebugUnitTest - Run unit tests for the debug build.
testReleaseUnitTest - Run unit tests for the release build.

2.1 By families

At first, it's a bit confusing. But in fact there is a lot of tasks specified by build variants/type/context, if we have a look by big families (I remove the help and other family) it can be resumed:

All tasks runnable from root project

Android tasks

androidDependencies - Displays the Android dependencies of the project.
signingReport - Displays the signing info for each variant.
sourceSets - Prints out all the source sets defined in this project.

Build tasks

assemble ***.
build ***
clean.
compile**

Install tasks

Install*** Install all applications.
Uninstall***

Verification tasks

check.
connected***
device***
lint***
test***

Better vision isn't it, so let's look at them.

2.2 Android tasks family

We have 3 really cool tasks here (especially when you play with flavors):

2.2.1 androidDependencies

Gradlew androidDependencies will show you, variant by variant, what libraries dependencies graph is.

```

D:\Git\MyProjets\MultiplicationBasile>gradlew androidDependencies
The JavaCompile.setDependencyCacheDir() method has been deprecated and is scheduled to be removed in Gradle 9.0.
Incremental java compilation is an incubating feature.
The TaskInputs.source(Object) method has been deprecated and is scheduled to be removed in Gradle 9.0.
:app:androidDependencies
debug
\--- com.android.support:appcompat-v7:25.1.0
| +--- com.android.support:support-v4:25.1.0
| | +--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-media-compat:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-utils:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-ui:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | \--- com.android.support:support-fragment:25.1.0
| | +--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-media-compat:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-ui:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | \--- com.android.support:support-core-utils:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| +--- com.android.support:support-vector-drawable:25.1.0
| | \--- com.android.support:support-compat:25.1.0
| \--- com.android.support:animated-vector-drawable:25.1.0
| | \--- com.android.support:support-vector-drawable:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
debugAndroidTest
\--- com.android.support.test.espresso:espresso-core:2.2.2
| +--- com.android.support.test:rules:0.5
| | \--- com.android.support.test:runner:0.5
| | | \--- com.android.support.test:exposed-instrumentation-api-publish:0.5
| +--- com.android.support.test:runner:0.5
| | \--- com.android.support.test:exposed-instrumentation-api-publish:0.5
| \--- com.android.support.test.espresso:espresso-idling-resource:2.2.2
debugUnitTest
No dependencies
release
\--- com.android.support:appcompat-v7:25.1.0
| +--- com.android.support:support-v4:25.1.0
| | +--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-media-compat:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-utils:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-ui:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | \--- com.android.support:support-fragment:25.1.0
| | +--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-media-compat:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | +--- com.android.support:support-core-ui:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| | \--- com.android.support:support-core-utils:25.1.0
| | | \--- com.android.support:support-compat:25.1.0
| +--- com.android.support:support-vector-drawable:25.1.0
| | \--- com.android.support:support-compat:25.1.0
| \--- com.android.support:animated-vector-drawable:25.1.0
| | \--- com.android.support:support-vector-drawable:25.1.0
| | | \--- com.android.support:support-compat:25.1.0

```

2.2.2 Transitive dependencies

So it's the moment we talk about transitive dependencies. By the way, you shouldn't because gradle do it for you with smart algorithms. But sometimes we need to make it ourselves.

We can exclude transitive dependencies of a libraries using the transitive flag, like this:

```
compile ('com.android.support:appcompat-v7:25.1.0', {
    transitive= false
})
```

The effect is immediate, using a gradlew androidDependencies:

```
D:\Git\MyProjets\MultiplicationBasile>gradlew androidDependencies
The JavaCompile.setDependencyCacheDir() method has been deprecated and is scheduled to be removed in Gradle 4.0.
Incremental java compilation is an incubating feature.
The TaskInputs.source(Object) method has been deprecated and is scheduled to be removed in Gradle 4.0.
:app:androidDependencies
debug
\--- com.android.support:appcompat-v7:25.1.0

debugAndroidTest
\--- com.android.support.test.espresso:espresso-core:2.2.2
    +--- com.android.support.test:rules:0.5
         | \--- com.android.support.test:runner:0.5
         |      \--- com.android.support.test:exposed-instrumentation-api-publish:0.5
    +--- com.android.support.test:runner:0.5
         | \--- com.android.support.test:exposed-instrumentation-api-publish:0.5
         \--- com.android.support.test.espresso:espresso-idling-resource:2.2.2

debugUnitTest
No dependencies

release
\--- com.android.support:appcompat-v7:25.1.0

releaseUnitTest
No dependencies

BUILD SUCCESSFUL
```

We can also exclude specific dependencies from a library:

```
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2',
{
    exclude group: 'com.android.support', module: 'support-annotations'
})
```

It's what we do in our build.gradle by default for resolving conflict (by the way, I don't catch it).

2.2.3 signingReport

This task is really cool too, it show you what are the signing configuration of your project, which is a really important element when managing several flavor and build type.

So it shows you all your signature key like that:

```

:app:signingReport
Variant: release
Config: none
-----
Variant: debug
Config: debug
Store: C:\Users\mathi\.android\debug.keystore
Alias: AndroidDebugKey
MD5: F2:C0:42:E1:AE:A1:A9:63:16:26:79:D2:04:C6:A1:44
SHA1: 11:1D:16:C3:DE:AD:02:29:F4:23:93:1F:29:A5:F3:28:F1:25:57:FE
Valid until: dimanche 15 d'Ucembre 2041
-----
Variant: releaseUnitTest
Config: none
-----
Variant: debugAndroidTest
Config: debug
Store: C:\Users\mathi\.android\debug.keystore
Alias: AndroidDebugKey
MD5: F2:C0:42:E1:AE:A1:A9:63:16:26:79:D2:04:C6:A1:44
SHA1: 11:1D:16:C3:DE:AD:02:29:F4:23:93:1F:29:A5:F3:28:F1:25:57:FE
Valid until: dimanche 15 d'Ucembre 2041
-----
Variant: debugUnitTest
Config: debug
Store: C:\Users\mathi\.android\debug.keystore
Alias: AndroidDebugKey
MD5: F2:C0:42:E1:AE:A1:A9:63:16:26:79:D2:04:C6:A1:44
SHA1: 11:1D:16:C3:DE:AD:02:29:F4:23:93:1F:29:A5:F3:28:F1:25:57:FE
Valid until: dimanche 15 d'Ucembre 2041
-----

```

Which is definitely cool when your project have to register on api server like Google Api. All the sha1 you need to have are just displayed here and in a glance you see how many keys you need to register. The magic trick.

2.2.4 sourceSets

This one show you for each variant where are the information of your project. What is the task for compiling this variant, where are the sources (java,manifest,res,assets,...).

So it gives you a good vision of your project's variants parameters.

```
-----
Project :app
-----

androidTest
-----
Compile configuration: androidTestCompile
build.gradle name: android.sourceSets.androidTest
Java sources: [app\src\androidTest\java]
Manifest file: app\src\androidTest\AndroidManifest.xml
Android resources: [app\src\androidTest\res]
Assets: [app\src\androidTest\assets]
AIDL sources: [app\src\androidTest\aidl]
RenderScript sources: [app\src\androidTest\rs]
JNI sources: [app\src\androidTest\jni]
JNI libraries: [app\src\androidTest\jniLibs]
Java-style resources: [app\src\androidTest\resources]

debug
-----
Compile configuration: debugCompile
build.gradle name: android.sourceSets.debug
Java sources: [app\src\debug\java]
Manifest file: app\src\debug\AndroidManifest.xml
Android resources: [app\src\debug\res]
Assets: [app\src\debug\assets]
AIDL sources: [app\src\debug\aidl]
RenderScript sources: [app\src\debug\rs]
JNI sources: [app\src\debug\jni]
JNI libraries: [app\src\debug\jniLibs]
Java-style resources: [app\src\debug\resources]

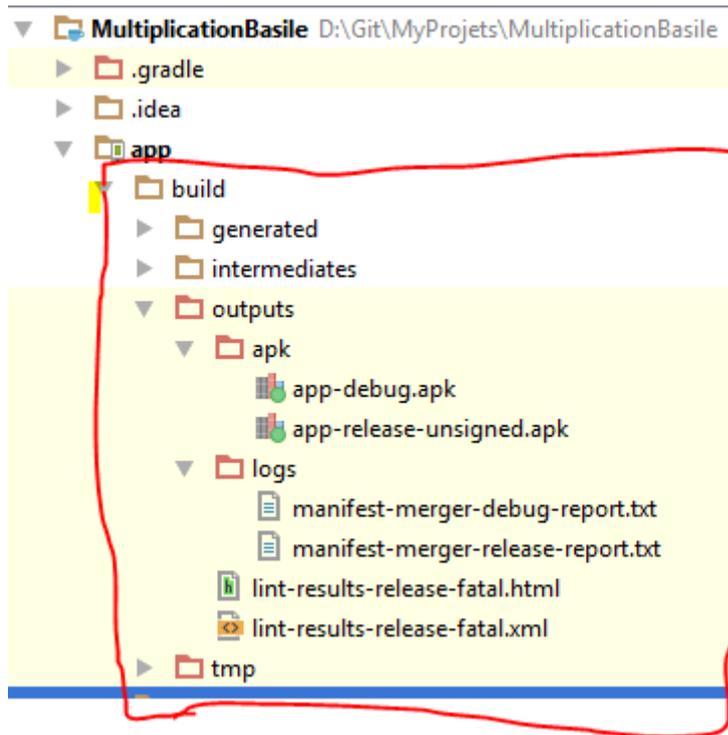
main
-----
Compile configuration: compile
build.gradle name: android.sourceSets.main
Java sources: [app\src\main\java]
Manifest file: app\src\main\AndroidManifest.xml
Android resources: [app\src\main\res]
Assets: [app\src\main\assets]
AIDL sources: [app\src\main\aidl]
RenderScript sources: [app\src\main\rs]
JNI sources: [app\src\main\jni]
```

2.3 Build tasks family

In this family we have assemble^{***}, build^{***} and compile^{***}. Each of them has a declination depending on the variants (Debug/AndroidTest/Release...) they apply on. Those declination depends on the task.

2.3.1 assemble:

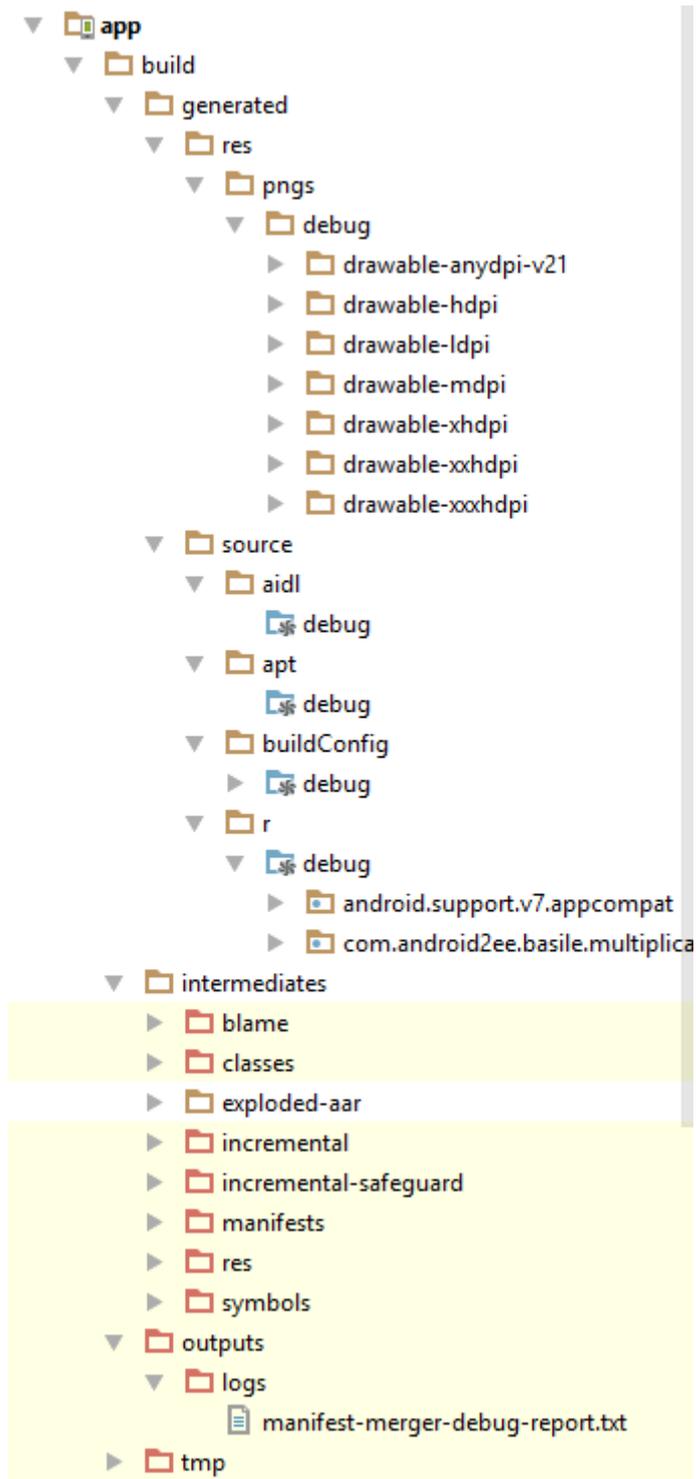
It will create your apk and your report (lint and merge) for all your variants.



AssembleDebug will do the same but only for the debug variant and so on for ***Release...

2.3.2 compile

It launch the "compilation" of a specific set. Here compilation means prepare the files needed for creating the apk. For example compileDebugSources will manages your ressources for the debug.

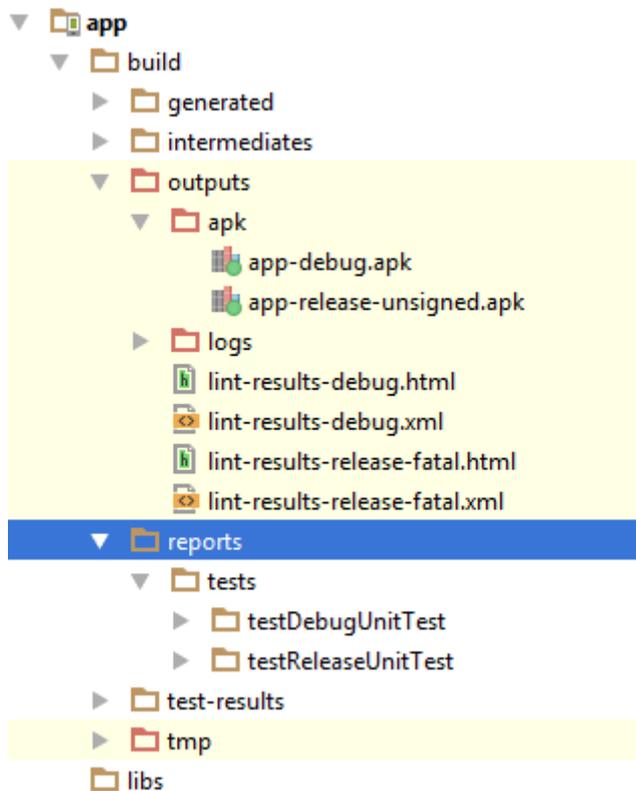


When assembling you always call all the different compile*** task.

2.3.3 build

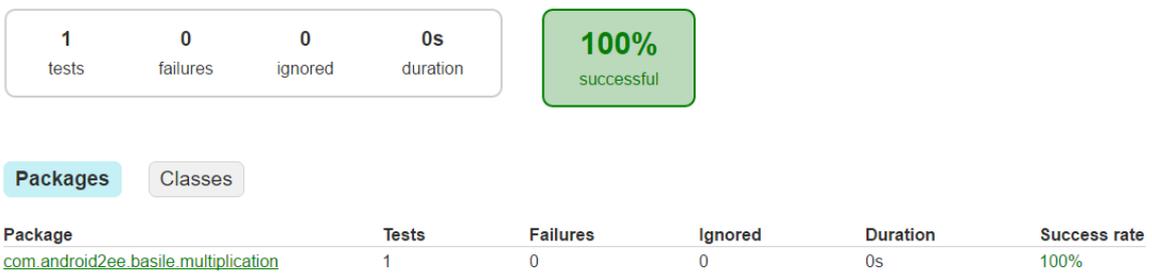
Build will assemble and test the project... Test the project ?o? In fact it will run your Unit tests not your instrumentation tests. We will see that we have specific tasks for instrumentation tests. So don't be too confident on build.

By the way, you have all the reports of your unit tests and lint generated and also the apk:



If you open a test report you'll have a web site describing your tests, their results.

Test Summary



Generated by Gradle 3.3 at 2 mars 2017 22:01:13

The same occurs for lint reports.

2.4 Install tasks family

Those family assemble and then install specific variants on all connected devices or uninstall them.

```
:app:assembleDebug UP-TO-DATE
```

```
:app:installDebug
```

```
Installing APK 'app-debug.apk' on 'Nexus 5X - 7.1.1' for app:debug
```

```
Installed on 1 device.
```

```
BUILD SUCCESSFUL
```

2.5 Verification task family

2.5.1 check

This task run all the checks... blurry description in fact. This task runs all Unit test for all variants and also generates Lint reports. But still doesn't run your instrumentation tests.

2.5.2 clean

This task is simple, it just delete the build folder.

2.5.3 connected**

Here comes your instrumentation test.

The task `connectedAndroidTest` installs and runs your instrumentations tests for all flavors on all connected devices :o) Yes we have it.

The task `connectedCheck` runs your unit tests on connected device (why? Tell me why running unit test on devices or emulator?)

The task `connectedDebugAndroidTest` installs and runs your instrumentations tests for debug variants on all connected devices.

2.5.4 lint***

Run lint on all or specific flavors.

2.5.5 Test***

Run unit tests on all or specific flavors.

2.6 Others details

2.6.1 Creating the fullBuild task

We saw that the build task is only running assemble and check. What if we want a fullBuild that runs clean assemble check and `connectedAndroidTest` ?

Let's do it.

If you want to define a task, you do it for your whole project, in root project `build.gradle` :

// Top-level build file where you can add configuration options common to all sub-projects/modules.

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.3'
        classpath 'io.balto:balto-plugin:2.0.2'
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

```
task clean(type: Delete) {
    delete rootProject.buildDir
}
```

```
task fullBuild(dependsOn: [':app:build','app:connectedAndroidTest']) {
    doLast {
        println 'Done'
    }
}
```

Ok, it's done. I have defined the task fullBuild that depends on the task build and connectedAndroidTest of my module app. So when I will run it, it will run build and then connectedAndroidTest. So I will have a real build with all my tests done (if I plug a device or an emulator). This can be usefull when you play with Jenkins.

2.6.2 Android Gradle DSL

To go further you can see the description of the Android Gradle DSL and use it:

<https://github.com/google/android-gradle-dsl>

And you have the Gradle SDK (in a way) that you can use to customize task and know their parameters.

2.6.3 Build profiling

Sometimes your build takes too long and you want to profile it, just add --profile and you have the time report of your build (in rootproject/build/report/profile/)

Profile report

Profiled build: assemble

Started on: 2017/03/02 - 23:06:36

Summary

Configuration

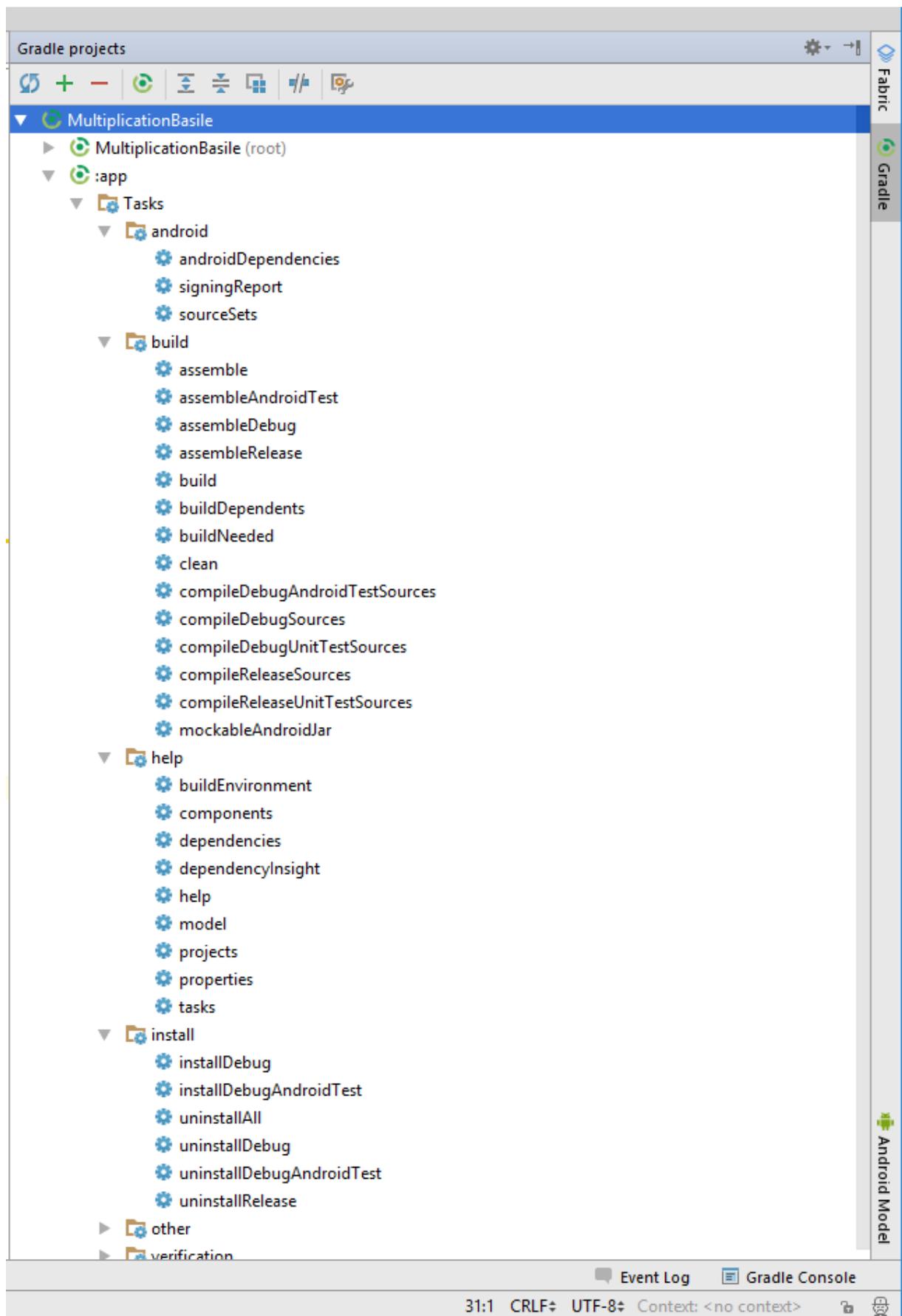
Dependency Resolution

Task Execution

Description	Duration
Total Build Time	7.299s
Startup	1.001s
Settings and BuildSrc	0s
Loading Projects	0s
Configuring Projects	0.360s
Task Execution	5.321s

3 Simpler with AndroidStudio

I can do all that with AndroidStudio by clicking on the specific task I want to run. It's hidden in the right bar:



4 Conclusion

When I finished to write this article I was, “Ok dude, there are the android tasks, now I know them, but I don’t use them and everything can be done using AndroidStudio in a better way”. Yep, an unnecessary article in a way but perhaps, I make the first step to understand tasks and perhaps I it will be useful when trying to make deployment, delivery or continuous integration. I will see but I didn’t found anything that will help me in my daily coding.

Chapter3: Flavors, BuildTypes and Variants

Let's keep discovering Gradle and now it's time to talk about flavors, build types and variants. Yes. All the discussion here belongs to your module (not the project) build.gradle.

Flavors is the ability to overwrite files at compilation time. We have a lot of tricks to adapt our code behavior depending on context (connectivity, battery, lowram device, gingerbread) using interfaces and factories. But until now, it was not possible to easily build several flavors of a same product and so adapt our product at compilation time.

And this magi is Flavors. But let's begin by the beginning and let's have a look at the build type notion.

5 BuildTypes

By default there are two build types: the one for debug and the one for production.

It's not compile time adaptation because you don't release to the public a debug version. Build type are really part of the process development more than a customization of your application.

We generally have this bloc in our android block (of our build.gradle):

```
signingConfigs {
    release {
        storeFile file("javaKeyStore.jks")
        storePassword "passwordOfTheJks"
        keyAlias "keyName"
        keyPassword "passwordOfTheKey"
    }
}

buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        signingConfig signingConfigs.release
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
    }
    debug {
        applicationIdSuffix '.debug'
        versionNameSuffix '.debug'
    }
}
```

This is the most generic buildTypes you can have.

When we look at the release block, we have the signingConfig to sign the apk for the release. When have enabled shrink and minify to remove the unnecessary code and resources. We run proguard to obfuscate our code.

A word about proguard: It's a pain, especially when you have libraies in your project. A trick: Make often release build to see if something have changed don't wait the day before the release to production, you'll regret.

When we look at the debug blog, we have the suffix added. The one for the application is the most important, it give us the ability to deploy on the same device the both version (it's related to the unicity constraints on {applicationId,signatureKey}).

We'll see in another chapter how to extract your password and id into properties files, avoiding having them in the git repo or in the project (security reasons).

6 Flavors

Let's start with an example. I made a quick application for my son to learn multiplication tables.



And I said to myself, why don't I do one for my daughter too ? One for multiplication, the other for addition with specific branding for each of them.

But I don't want to copy paste the project and start a second one. Because I will lose all the improvements done on the other project. Complex branching system using git... No, too hazardous and full of merges. I could also have make a core library, but what a mess for a simple request:

I just want to customize pieces of my code, living the architecture at the center and customize the features according to a specific branding.

The answer is flavors.

When you compose **buildTypes** and **flavors**, it gives you build **variants**.

6.1 Basics

So I create it:

```
productFlavors{
    basile{
    }
    lila{
        applicationId "com.android2ee.lila.multiplication"
    }
}
```

And then, I run gradlew sourceSets and compare with what it was before. I remove content and keep only head chapter to gain in visibility:

<p>When I run sourceSets without flavor:</p> <ul style="list-style-type: none"> androidTest debug main release test testDebug testRelease 	<p>When I run sourceSets after creating the flavors:</p> <ul style="list-style-type: none"> androidTest androidTestBasile androidTestLila basile basileDebug basileRelease debug lila lilaDebug lilaRelease main release test testBasile testBasileDebug testBasileRelease testDebug testLila testLilaDebug testLilaRelease testRelease
--	--

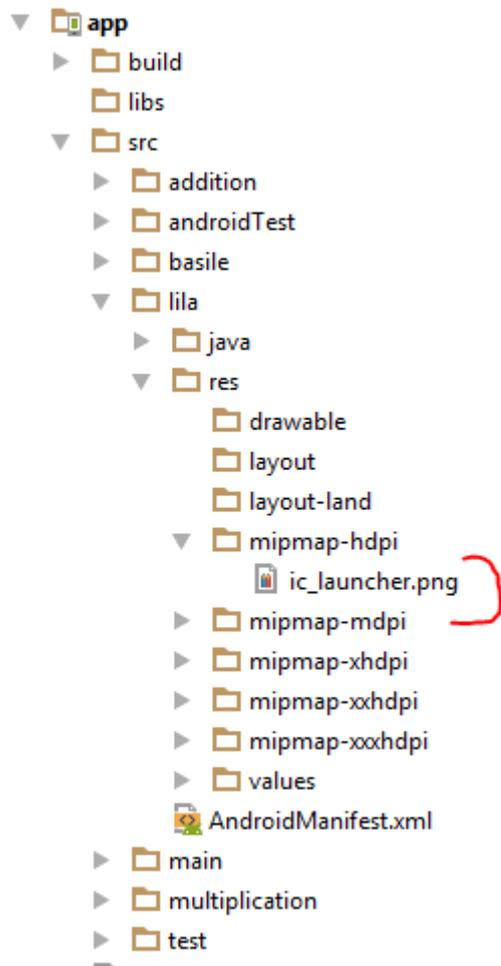
And yes, sourceSets shows you for each variant where are the information of your project. What is the task for compiling this variant, where are the sources (java,manifest,res,assets,...).

But another vision is sourceSets lists all the subfolders your can create in your application module to overwrite specific elements depending on a variants (flavor:Basile, build type:Release) and tests.

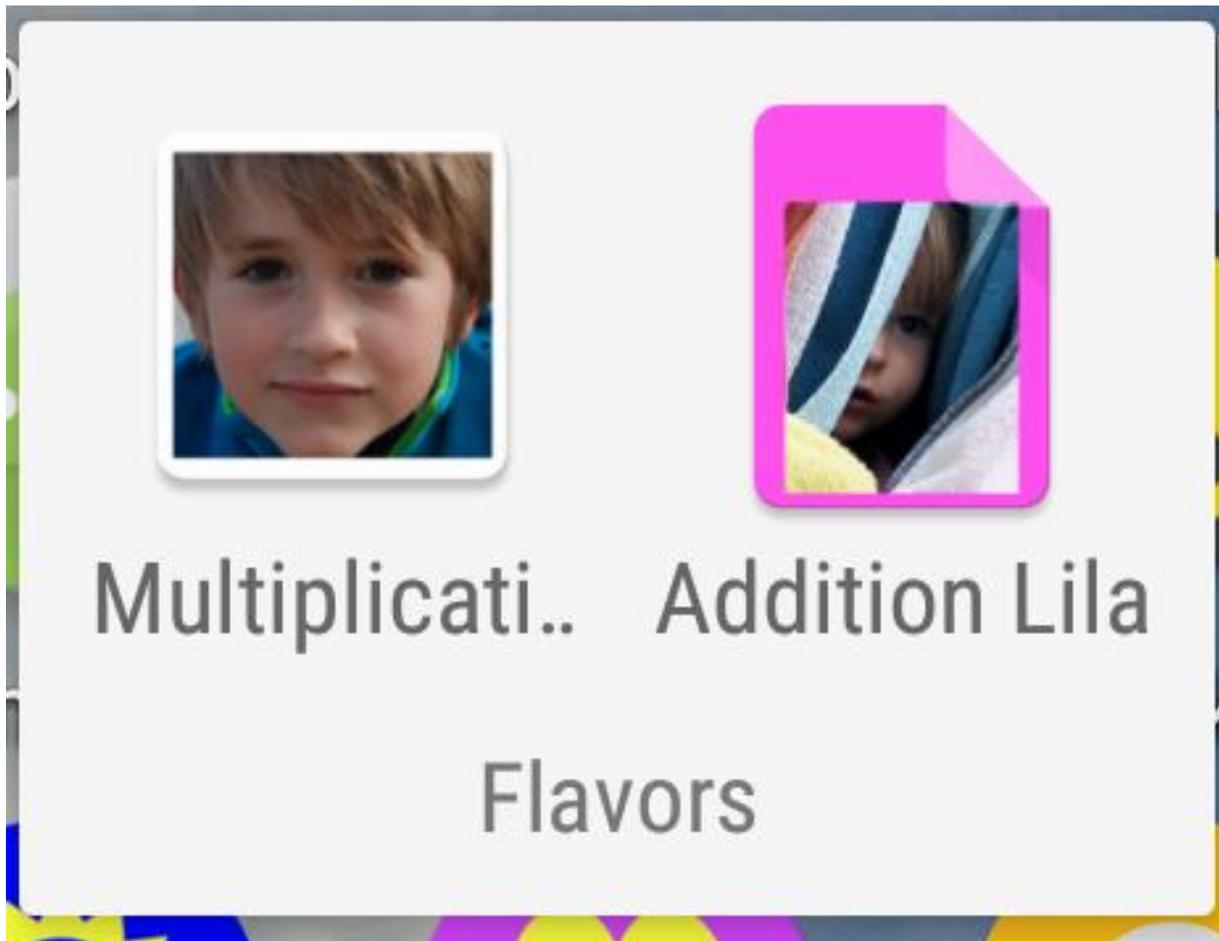
And that all. You want to overwrite a specific file, for exemple ic_launcher, because in Debug for Lila you want it with a smile, just define it under :

lilaDebug/res/mipmap-*/ic_launcher

If you want a specific for all lila variants, just do it in lila folder, and it's done:



I mean, it's done:



It works the same for all the resources and Java code. You just need to overwrite the file defined in main by your in the variants you want. It's based on the path of the file.

So flavors are ordered and it's important because it will defined how files are overwritten. For example, in BasileMultiplicationDebug, if the same element is defined in Basile and Multiplication, the one of Basile will be the one that stays, overwriting the one of multiplication. We'll see the flavor Multiplication latter).

The other important information to have here is:

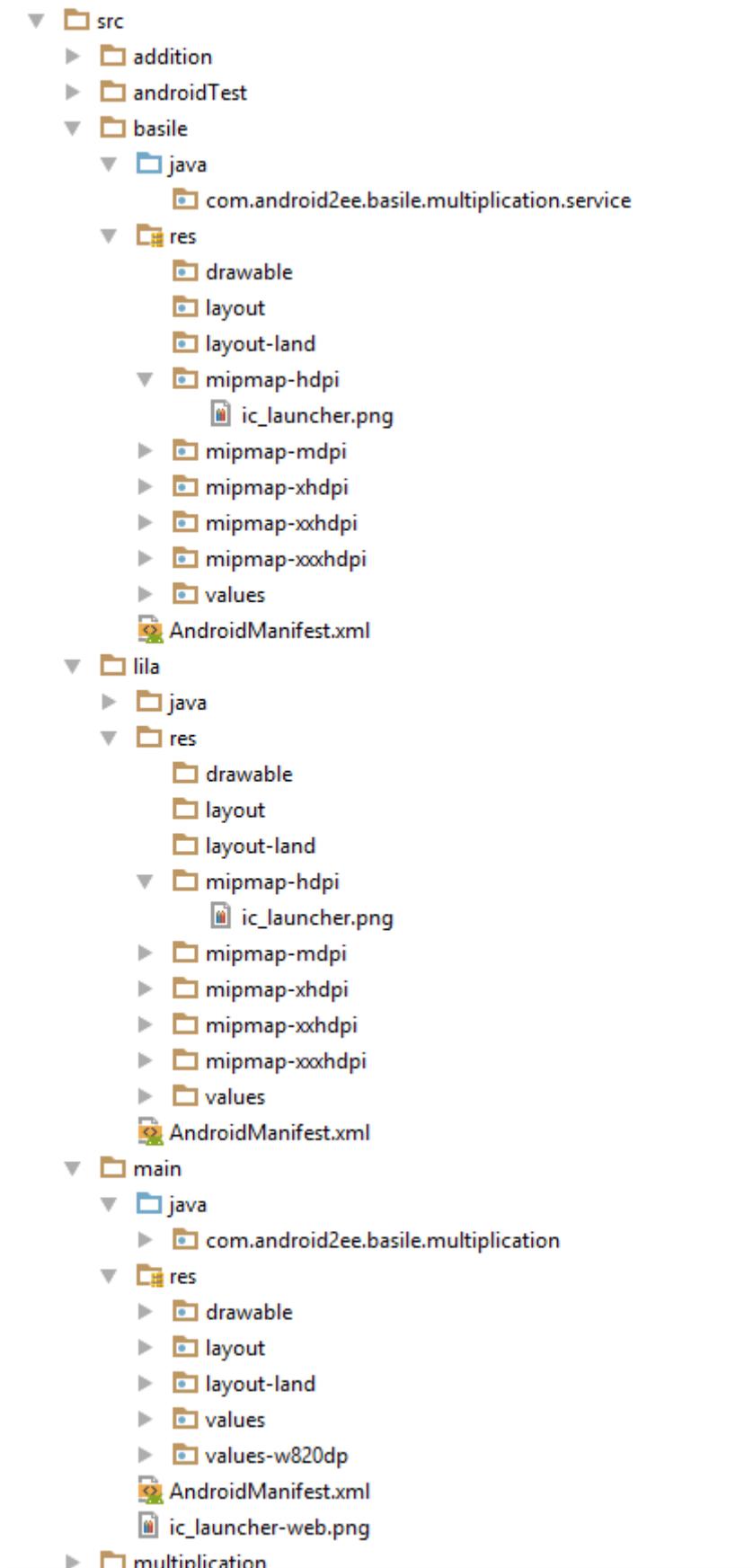
Flavor's resources can overwrite main's resources

Flavors' Java code CAN NOT overwrite those of main.

Most of time, on the project, you have mocked and prod flavors, where

- mocked is a flavor for test where all your services/dao/communication is mocked and dedicated to unit test/ integration tests
- prod is the flavor where you have your real application's bricks bound together

They belong to the same dimension "TestContext" or something like that.



6.2 Dimensions

Then I thought, ok, but Lila won't learn multiplication, she is too young, let's start with additions...
But wait, réflexion jean-pierre, one day, she will also make multiplications, so do I create two flavors instead of only one : lilaAddition and lilaMultiplication.

Yes, but no, I will think in terms of dimension :

	Addition	Multiplication
Lila	LilaAddition	LilaMultiplication
Basile	BasileAddition	BasileMultiplication

So I will say, I have two types of flavor, one is the kids and the other is operation. And if I could easily explain that basile doesn't need his addition dimension, it will be cool.

So let's go:

```
android {
    ...

    //Give a name to your dimension
    flavorDimensions "enfants", "operator"
    //define your flavors (as one flavor has a dimension they must all have
    one)
    productFlavors{
        basile{
            dimension "enfants"
        }
        lila{
            dimension "enfants"
            applicationId "com.android2ee.lila.multiplication"
        }
        multiplication{
            dimension "operator"
        }
        addition{
            dimension "operator"
        }
    }
    //Remove the BasileAddition and LilaMultiplication flavor
    android.variantFilter { variant ->
        if(variant.getFlavors().get(0).name.equals('basile')
            && variant.getFlavors().get(1).name.equals('addition')) {
            variant.setIgnore(true);
        }
        if(variant.getFlavors().get(0).name.equals('lila')
            && variant.getFlavors().get(1).name.equals('multiplication')) {
            variant.setIgnore(true);
        }
    }
}
```

Remember I have declared kids first, so the code and resources of the kids flavor will always overwrite those from operator and main.

In your flavor description you can define the following attributes:

- applicationId
- minSdkVersion

- targetSdkVersion
- versionCode
- versionName
- signingConfig

The last block is a little bit amazing, but we'll see that in another chapter more focus on task. The idea is to browse your flavors list and filter it based on flavor's name.

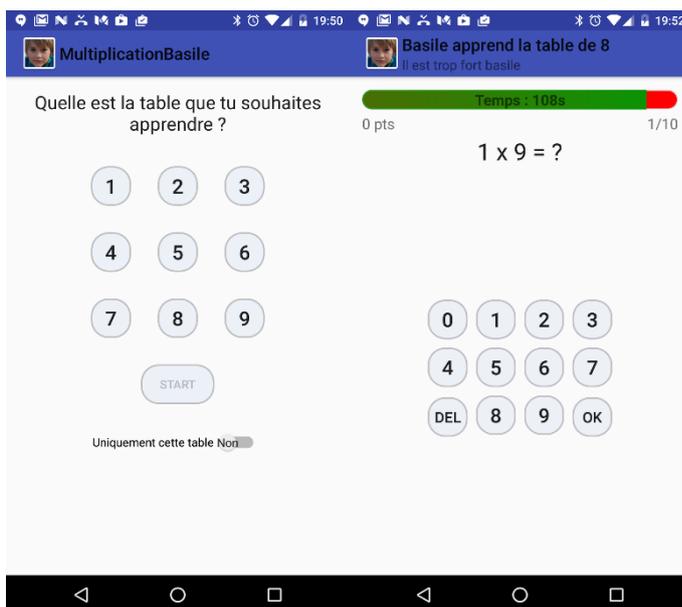
And I was also able to say, I will do it latter the LilaMultiplication one.

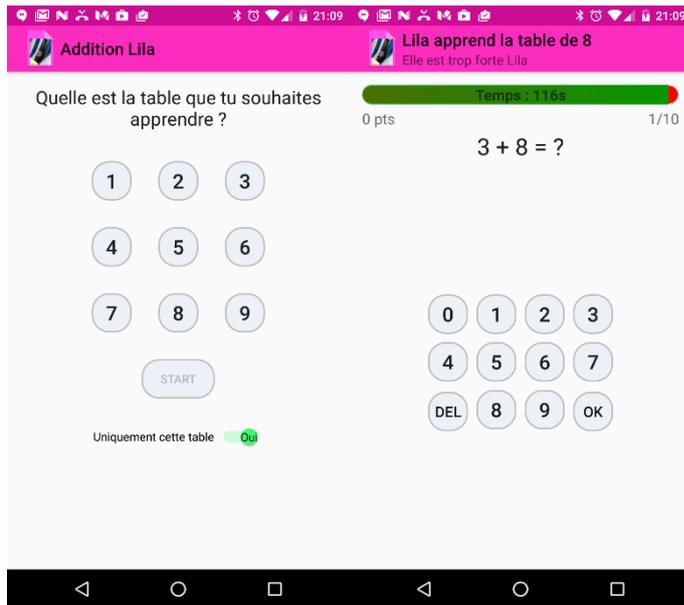
So when I run sourceSets, what happens ?

addition androidTest androidTestAddition androidTestBasile androidTestBasileMultiplication androidTestLila androidTestLilaAddition androidTestMultiplication basile basileMultiplication basileMultiplicationDebug basileMultiplicationRelease	debug lila lilaAddition lilaAdditionDebug lilaAdditionRelease main multiplication release test testAddition testBasile	testBasileMultiplication testBasileMultiplicationDebug testBasileMultiplicationRelease testDebug testLila testLilaAddition testLilaAdditionDebug testLilaAdditionRelease testMultiplication testRelease
---	--	--

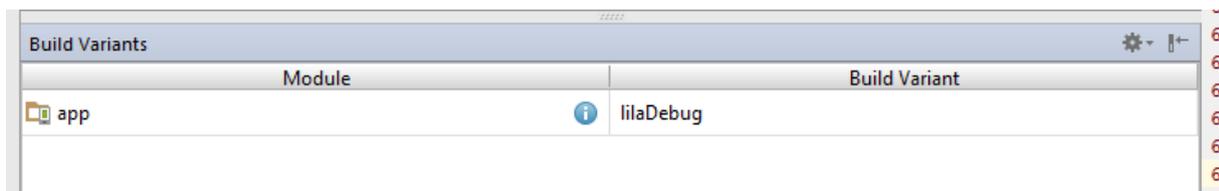
Et bien, ça en fait du monde. A lot of variants.

But at the ends, with only few changes and some copy/paste (yes, to create the flavor at first, copy/paste the folders from main to your flavor using the explorer, not in AS, to have the folders' structure), I achieve that:





And using AS, it's really easy, you have the build variants at the bottom right to help choose which variant you want to deploy:



7 Step by step example

7.1 Gradle file

Let's say I did my build.gradle file like explained:

```
android {
    ...

    //Give a name to your dimension
    flavorDimensions "enfants", "operator"
    //define your flavors (as one flavor has a dimension they must all have one)
    productFlavors {
        basile {
            dimension "enfants"
        }
        lila {
            dimension "enfants"
            applicationId "com.android2ee.lila.multiplication"
        }
        multiplication {
            dimension "operator"
        }
        addition {
            dimension "operator"
        }
    }
    //Remove the BasileAddition and LilaMultiplication flavor
    android.variantFilter { variant ->
```

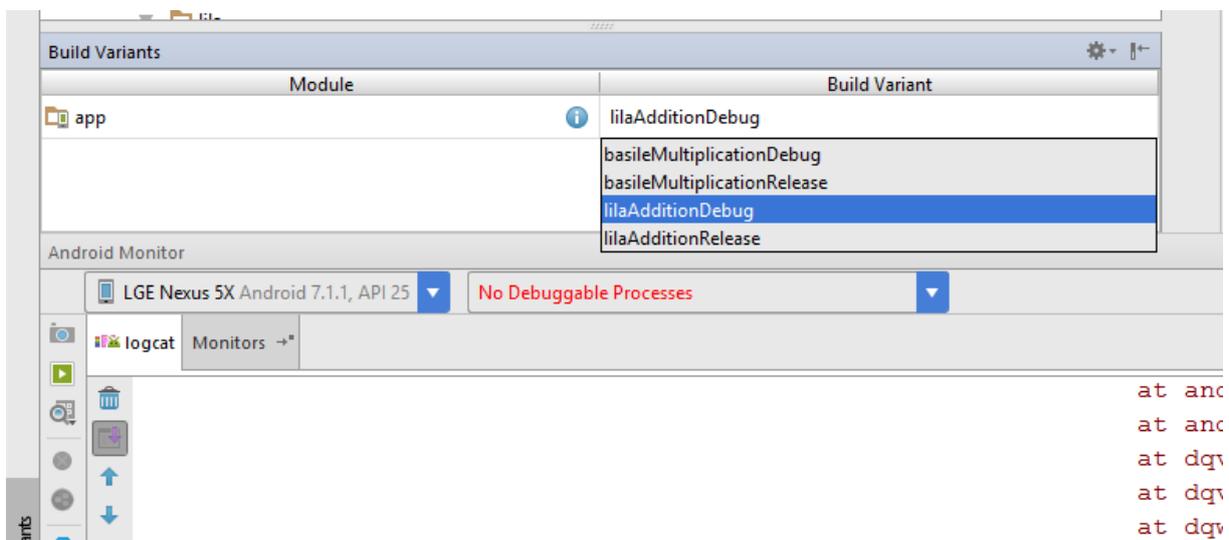
```

if (variant.getFlavors().get(0).name.equals('basile')
    && variant.getFlavors().get(1).name.equals('addition')) {
    variant.setIgnore(true);
}
if (variant.getFlavors().get(0).name.equals('lila')
    && variant.getFlavors().get(1).name.equals('multiplication')) {
    variant.setIgnore(true);
}
}
}

```

Note that with this gradle configuration I can deploy on the same device, Lila and Basile application because I change the applicationId for the Lila flavor.

Now, using AndroidStudio, build variants bar, I just have to select the build I want to deploy on the phone and click on the green arrow, like usual.



7.1.1 Custom task: print variants name

If I want to make a task that write the name of all the variants of the project. I have to define it in my build.gradle (always the same), like this :

```

task printVariantsName() {
    doLast {
        android.applicationVariants.all { variant ->
            println variant.name
        }
    }
}

```

So this code defined a *task* name *printVariantsName* and we that launched, when it has finished the "job", an action (real word is closure). This action just ask the android plugin to give us all its variants (*android.applicationVariants.all*). We take each variant one by one and print its name (*variant -> println variant.name*).

If I run it I have the following outputs:

lilaAdditionDebug

lilaAdditionRelease

basileMultiplicationDebug

basileMultiplicationRelease

Damned it, all this job to have the same result than in AndroidStudio with one click on a bar:)

7.2 Setting your folders structure

Ok, from my point of view, the best way and quickest way to create your folders structure is to use the files explorer or your system.

You first create your flavors folder. Then you copy paste src and res from main to all the flavors you created. And then **delete all the files** that are not folders in your flavors (be smart do it at the first paste...).

You're structure is finished you can go to work.

7.3 Customizing Resources

I want to customize strings, colors and icon launcher picture. All those elements are resources, so I have to copy paste them in lila and basile and adapt them as desired.

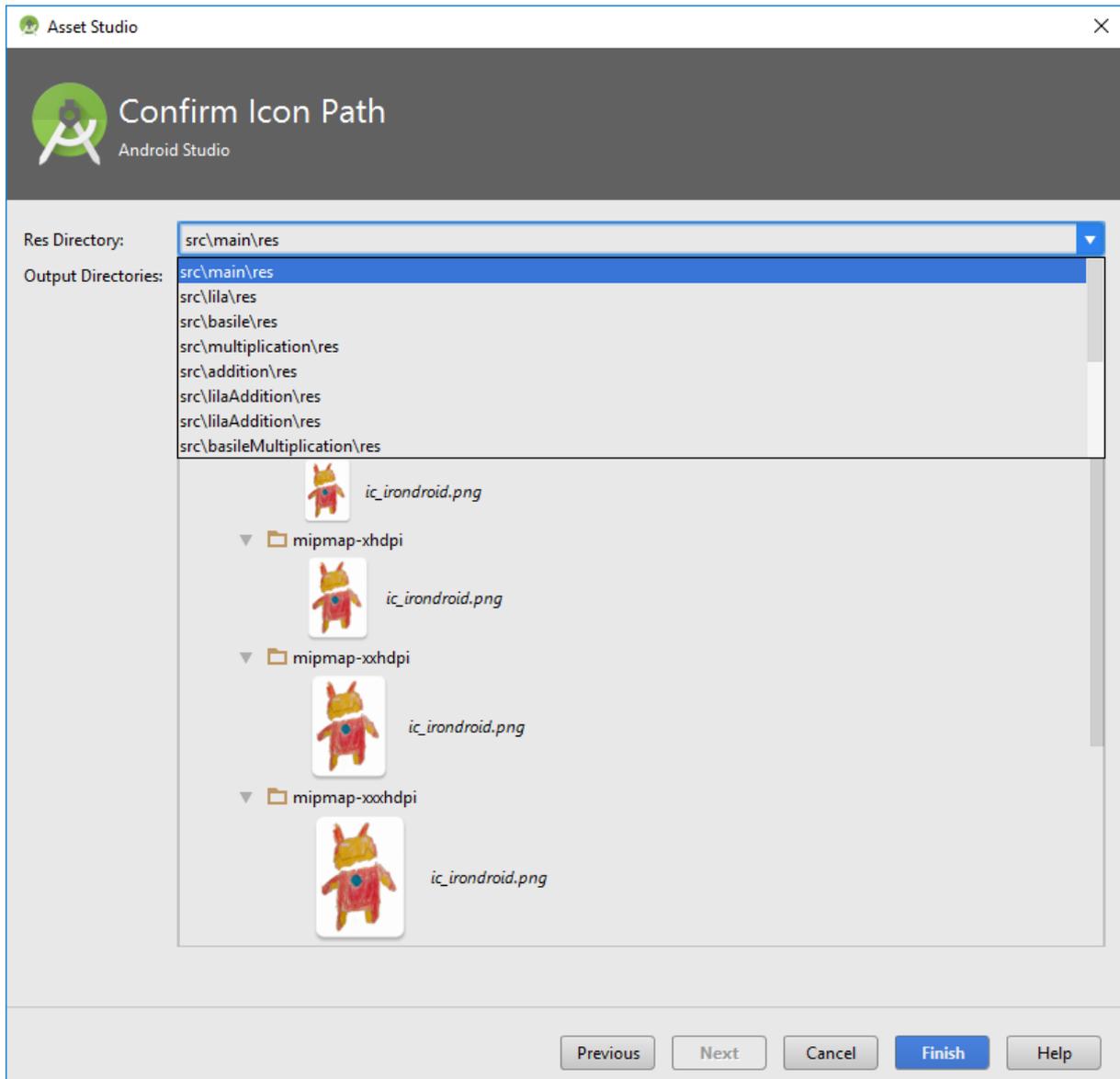
As I have no customization related to tests or instrumentation tests, I don't have to create more folders than my flavors declared. But if you want to customize a tests context, you can do it by creating the associated variant folder (like when running sourceSets).

A really good practice is to overwrite (copy files) only the resources you overwrite (really change). It means, don't copy by default all the files from main to yours flavors, do it only for files you change. I spent a quarter, adding a switch in a layout with no result because I had copied/pasted this layer also in the basile's flavor without paying attention.

7.3.1 Drawables customization

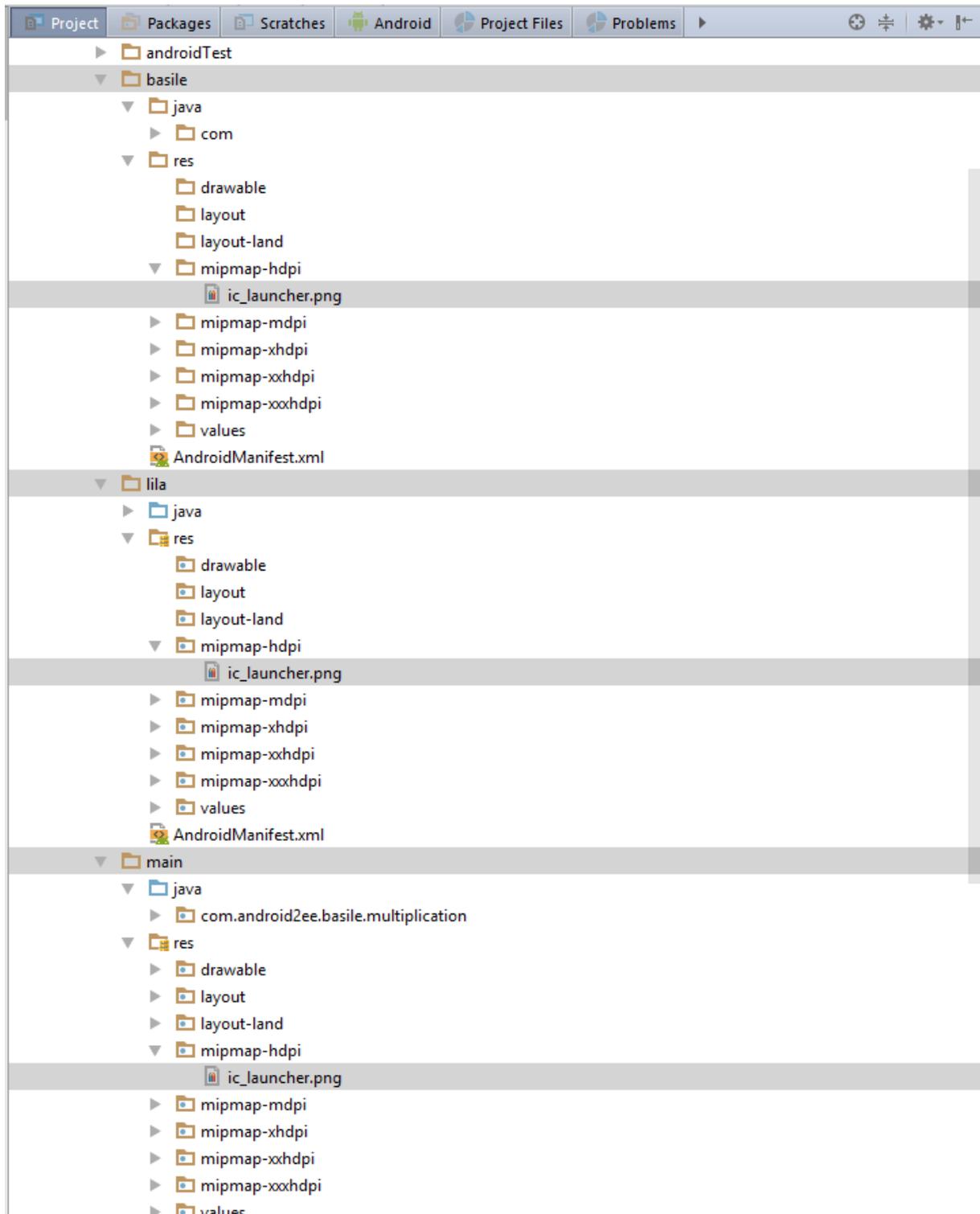
Let's begin by changing the icon of the application. For that the only stuff to do is to overwrite the ic_launcher (*dpi) in Lila and Basile flavors. To do that, just use the Android Studio Image creation wizard and create your ic_launcher for each flavor.

The last screen of the wizard with the button finish has a combo at its top where you define the flavor for this picture. Exactly what we need:



So you just have to create your mipmap and it's done.

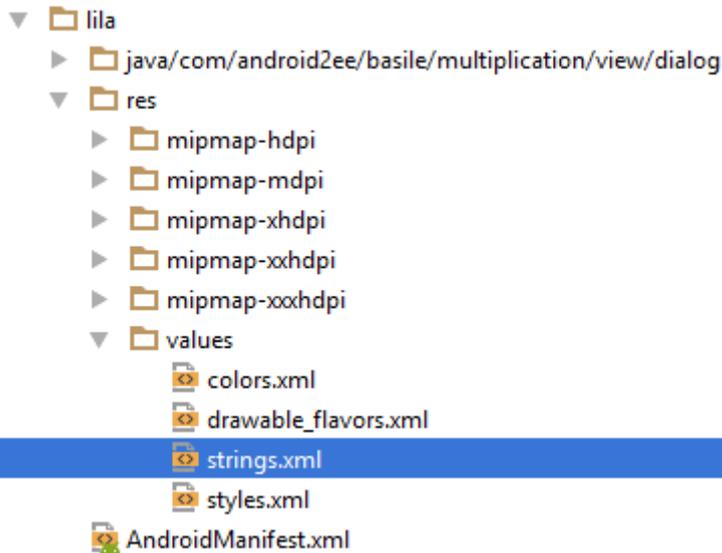
This is the result in my folder's structure:



7.3.2 String Customization

For the string, I did the same, but this time I copied/pasted the files instead of using a wizard and I delete the strings I didn't change:

lila/res/values/string.xml

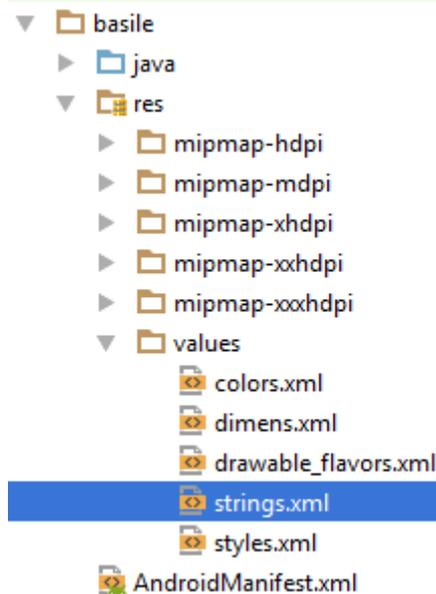


```

<resources>
  <string name="app_name">Addition Lila</string>
  <string name="ass_act_toolbar_title">Lila apprend la table de
%1$d</string>
  <string name="ass_act_toolbar_subtitle">Elle est trop forte
Lila</string>
  <string name="question_string">%1$d + %2$d = %3$d</string>
  <string name="question_string_init">%1$d + %2$d = ? </string>
</resources>

```

basile/res/values/string.xml



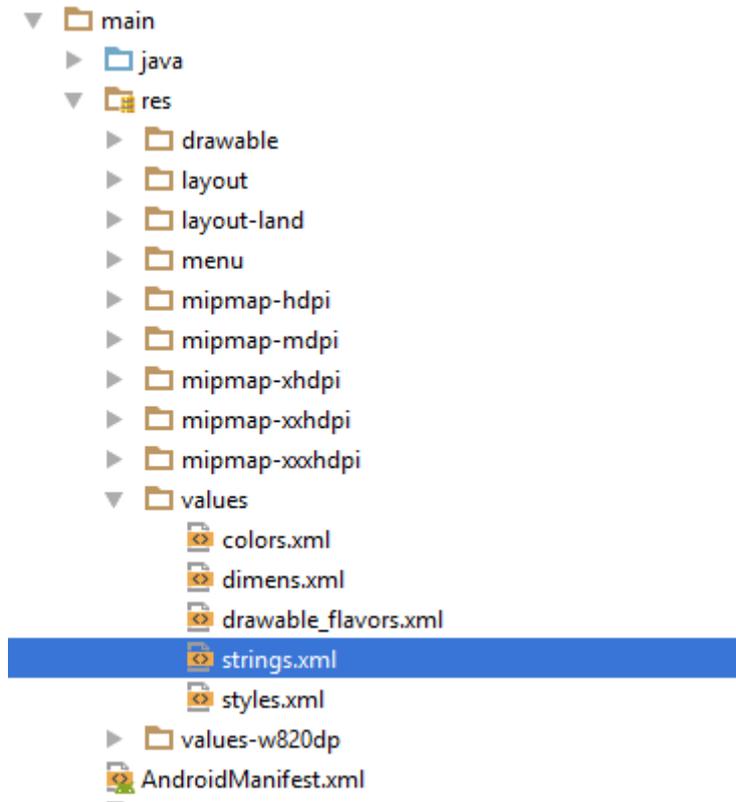
```

<resources>
  <string name="app_name">MultiplicationBasile</string>
  <string name="ass_act_toolbar_title">Basile apprend la table de
%1$d</string>
  <string name="ass_act_toolbar_subtitle">Il est trop fort
basile</string>
  <string name="question_string">%1$d x %2$d = %3$d</string>
  <string name="question_string_init">%1$d x %2$d = ? </string>

```

</resources>

main/res/values/string.xml



```
<resources>
  <string name="app_name">Default Title</string>
  <string name="ass_act_toolbar_title">Basile apprend la table de
%1$d</string>
  <string name="ass_act_toolbar_subtitle">Il est trop fort
basile</string>
  <string name="question_string">%1$d x %2$d = %3$d</string>
  <string name="question_string_init">%1$d x %2$d = ? </string>
  <string name="mainact_max_multiplication_value">Quelle est la table que
tu souhaites apprendre ?</string>
  <string name="mainact_maxvalue_edt_hint">Tape un nombre</string>
  <string name="mainact_txvTemps">Temps : %1$ds</string>
  <string name="mainact_txvScore">%1$d pts</string>
  <string name="mainact_txvQuestionNumber">%1$d/%2$d</string>
  <string name="mainact_answer_edt_hint"> \? </string>
  <string name="mainact_start">Start</string>
  <string name="mainact_switch_positive">Oui</string>
  <string name="mainact_switch_negative">Non</string>
  <string name="mainact_switch_question">Uniquement cette table</string>
</resources>
```

The important element to pay attention here is the string.xml of lila and basile flavors contain only strings that need to be adapted to the flavor. You don't have to copy all default strings. It's exactly

the same with you res folder drawable-hdpi/-mdpi-ldpi... The system merge the resources in a natural way. With flavor, always take attention to duplication and avoid it if unnecessary. Because,

With flavors when mess comes, mess is huge.

You can or not delete the string in main/string.xml already defined in all the flavors, but it's not as obvious as it looks first that the good practice. That's why I let them in my string.xml file. I prefer string error than NPE and I am not sure that this string has to be translated for all the flavors.

7.3.3 Colors customization

Then I want to specify colors to use for each flavor. So I copy my main\res\values\colors.xml to lila\res\values\colors.xml and basile\res\values\colors.xml. I change the values I want, especially the color Primary, PrimaryDark and Accent, because I rely on the support library.

lila\res\values\colors.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <color name="colorPrimary">#fc2cbe</color>
  <color name="colorPrimaryDark">#cc0c93</color>
  <color name="colorAccent">#40ff70</color>
  <color name="shape_default_stroke">#d4d4d4</color>
  <color name="shape_default_solid">#d4d4d4</color>
  <color name="item_background_translucent">#999e9c9c</color>
  <color name="item_background_opaque">#5fd5def2</color>
</resources>
```

basile\res\values\colors.xml

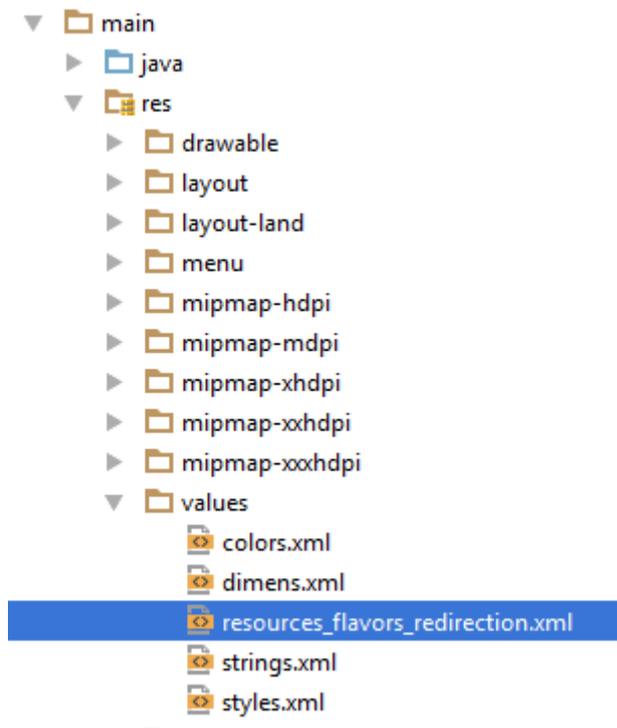
```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#0db656</color>
  <color name="shape_default_stroke">#d4d4d4</color>
  <color name="shape_default_solid">#d4d4d4</color>
  <color name="item_background_translucent">#999e9c9c</color>
  <color name="item_background_opaque">#5fd5def2</color>
</resources>
```

7.3.4 Resources Redirection

Sometimes, it's useful to have some concept on resources redirection. The typical example is you use a resource A which is used by a resource D, but D is also use for another resource C and you want to customize D within A but not within C. For example, D is a drawable, A is a layout (D is used as background) and C is an ImageView (and D is used as android:src or through code with setImageResource).

The simple way is to duplicate D and create DWithinA and DWithinC and customize DWithinA in your flavors. But, if D is drawable, you increase the size of your apk and the number of files in your project which is not a good idea.

Let's assume, D is a drawable. I make a redirection to it, I create a file called resources_flavor_redirection.xml in main:



And add the redirection inside:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <drawable
name="main_act_keyboard_background">@mipmap/ic_drhulk</drawable>
</resources>
```

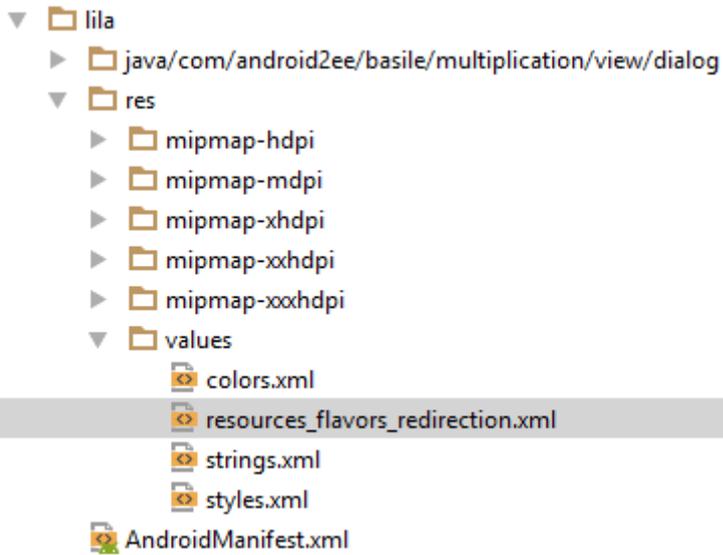
Then in A, which is in the example, the “background” of a layout in my main_activity.xml layout:

```
...<FrameLayout
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:foregroundGravity="center">
  <ImageView
    android:layout_width="285dp"
    android:layout_height="285dp"
    android:src="@drawable/main_act_keyboard_background"/>

    <include layout="@layout/keyboard"
      android:visibility="visible" />
</FrameLayout>...
```

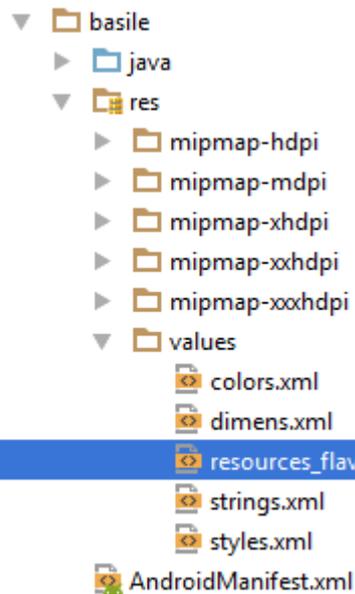
Then I just have to play the flavors game with my resources_flavor_redirection :

lila\res\values\resources_flavor_redirection.xml



```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <drawable
name="main_act_keyboard_background">@mipmap/ic_spiderdroid</drawable>
</resources>
```

basile\res\values\resources_flavor_redirection.xml



```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <drawable
name="main_act_keyboard_background">@mipmap/ic_captaindroid</drawable>
</resources>
```

And for the C component, nothing have changed. I can still use the initial D drawable without risk of missed overwriting issues, like in that line of code:

```

//then create you Dialog itself
builder.setIcon(R.mipmap.ic_captainandroid)
// Set Dialog Title
.setTitle("Game Over")
// Set Dialog Message
//.setMessage("Alert DialogFragment Tutorial")
//or your specfic view
.setView(view)

```

7.3.5 Declaring Resources in your gradle's flavor block

We can define attribute in our flavor block (and in our gradle file by the way).

For example:

```

defaultConfig {
    resValue "string", "hidden_string", "I love you my sweety"
}
//Give a name to your dimension
flavorDimensions "enfants", "operator"
//define your flavors
//(as one flavor has a dimension they must all have one)
productFlavors{
    basile{
        dimension "enfants"
        //resValue "boolean", "basile_dimension", "true" -> not allowed
        //resValue "int", "int_allowed", "1" -> not allowed
        resValue "string", "hidden_string", "I love you my basilou"
        resValue "color", "int_allowed", "#FF00ff"
    }
    lila{
        dimension "enfants"
        applicationId "com.android2ee.lila.multiplication"
        resValue "string", " hidden_string ", "I love you my Lila"
        resValue "color", "int_allowed", "#FF00ff"
    }
    multiplication{
        dimension "operator"
    }
    addition{
        dimension "operator"
    }
}
}

```

And you can use them in your code or your build or your manifest.

```

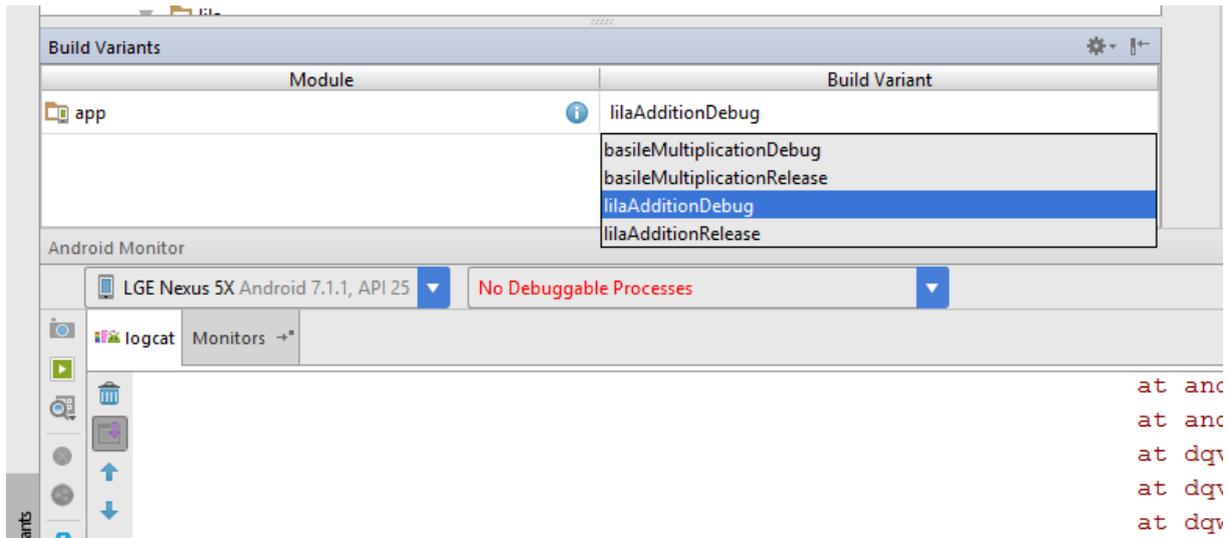
if(BuildConfig.isallowed) {
    Log.e(TAG, "BuldConfig IsAllowed and ResValue "
        +getString(R.string.hidden_string));
}

```

But I met problem with bool, int and others values, so I am a little disappointed.

7.3.6 Run your project(s)

It's over, our customization is finished, we can run the project. To do so, select the build you want and run it:

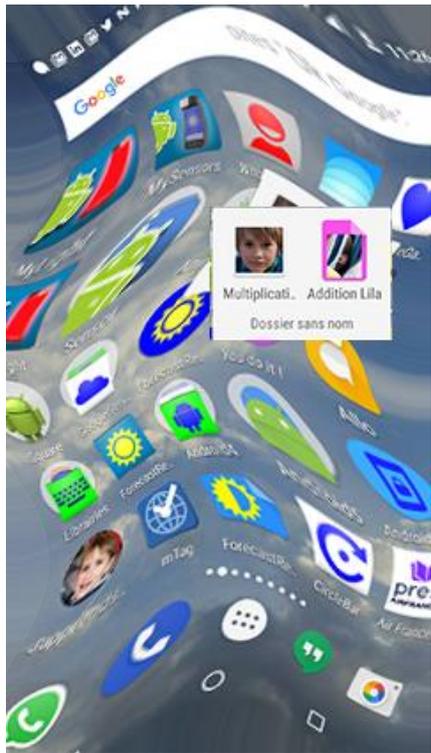


at anc
 at anc
 at dqv
 at dqv
 at dqv

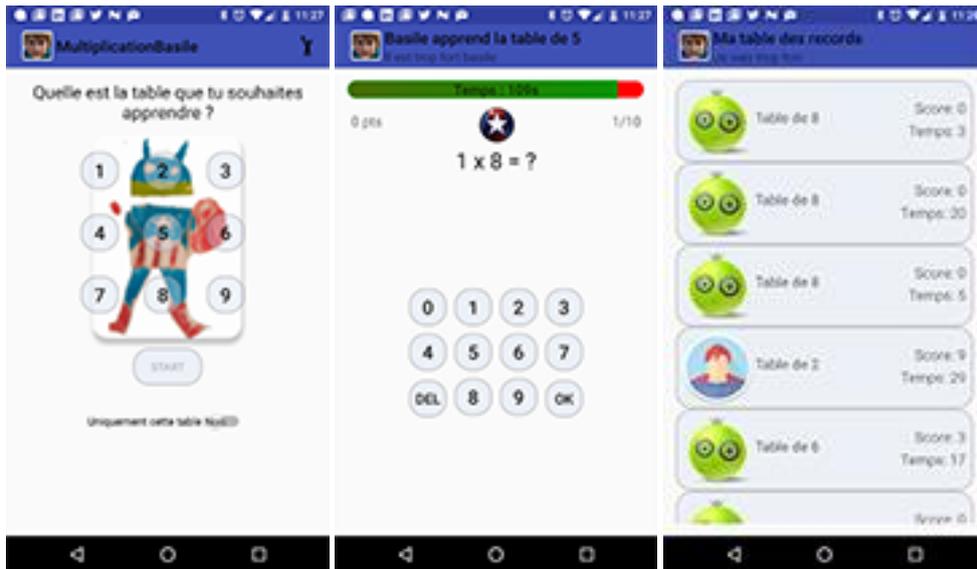
Tu peux aussi frimer en le faisant via gradlew install** si t'es en mode surfeur :)

You obtain:

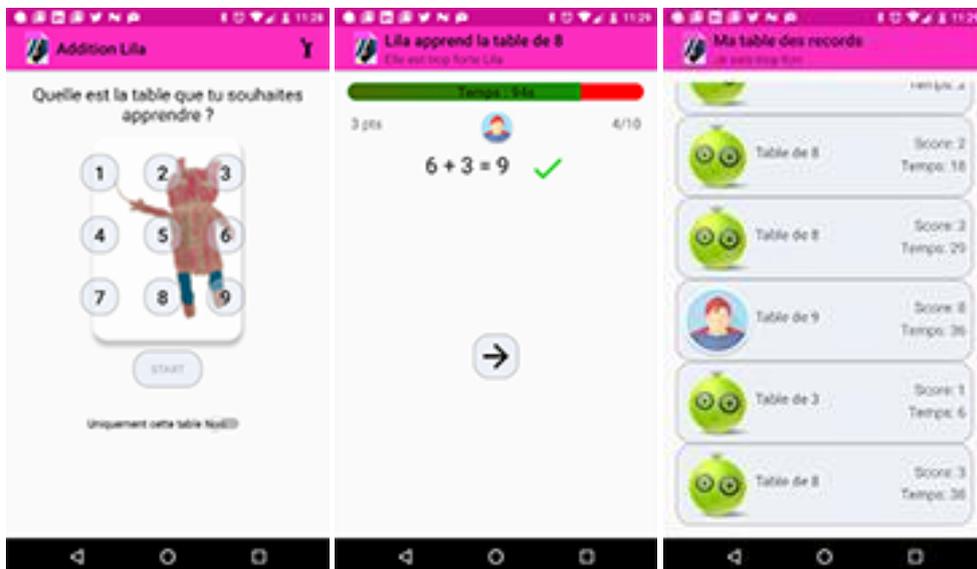
Both applications are installed and they have their own icon:



The Basile's application:



And the Lila's application:



7.4 Customizing Java code

The rule:

Flavor's resources can overwrite main's resources

Flavors' Java code CAN NOT overwrite those of main.

It means, in Java, you can not have a class overwritten for a specific flavor and have its default behavior defined in main. You need to remove the class from main and paste it in all your flavors. This force all others flavors to define this class. And it's a huge constraint.

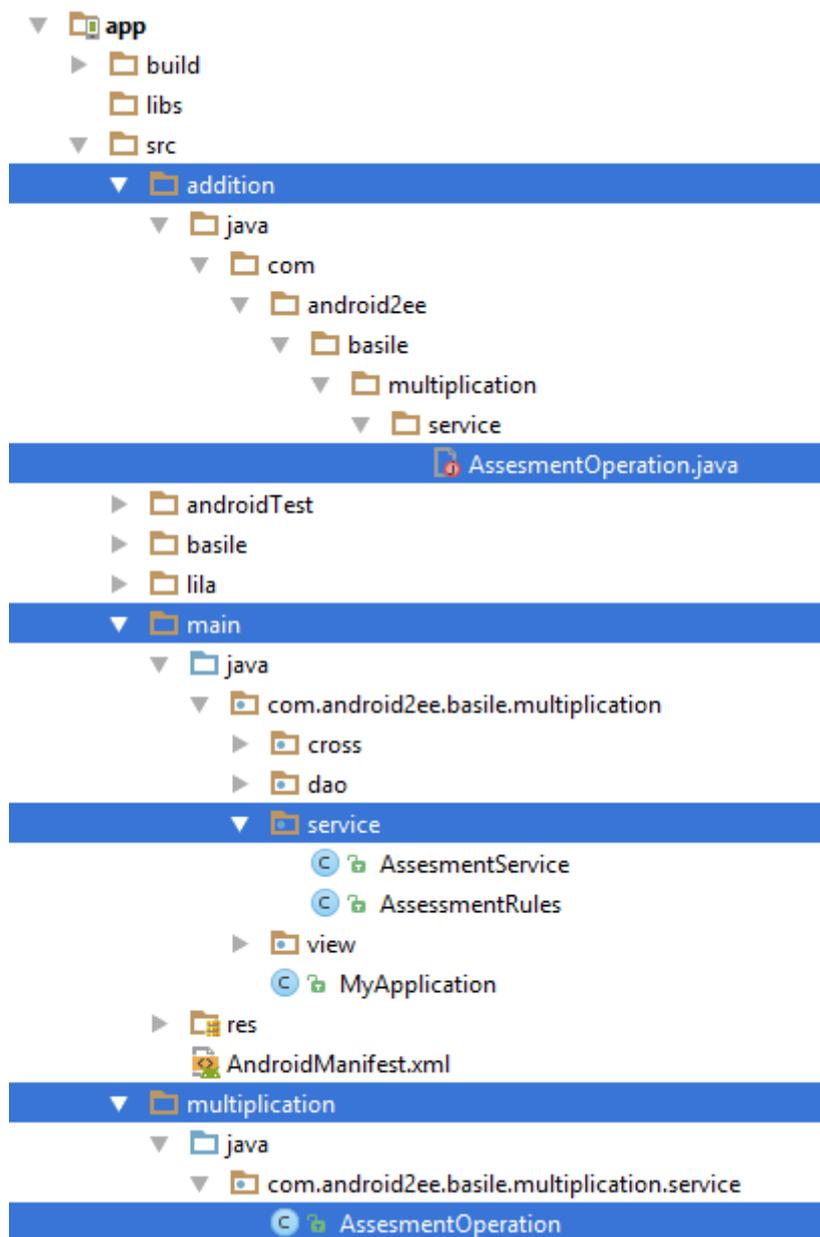
So, back to our example, one of my goal was to make a basile application to learn multiplication and a Lila one to make addition.

This is where Java code customization is needed in my project.

To do that, I just extract the operation * (everywhere a multiplication is done bound to the assessment process) from all my code and create a Java class called AssessmentOperation with one method `public int calculate(int value, int factor)`

That way, I extract the code I want to customize in a single small class. You will always have to refactor your code to make that isolation, else you will have too much code duplication.

So I create, in the flavor addition and multiplication (the ones in the “operator” dimension), my class AssessmentOperation. And I removed it from main:



then I just have to implement my method in each flavor (addition, multiplication):

multiplication\AssesmentOperation

```
public class AssesmentOperation {  
  
    public static int caculate(int value,int factor){  
        return value*factor;  
    }  
}
```

```
}  
}
```

addition\AssesmentOperation

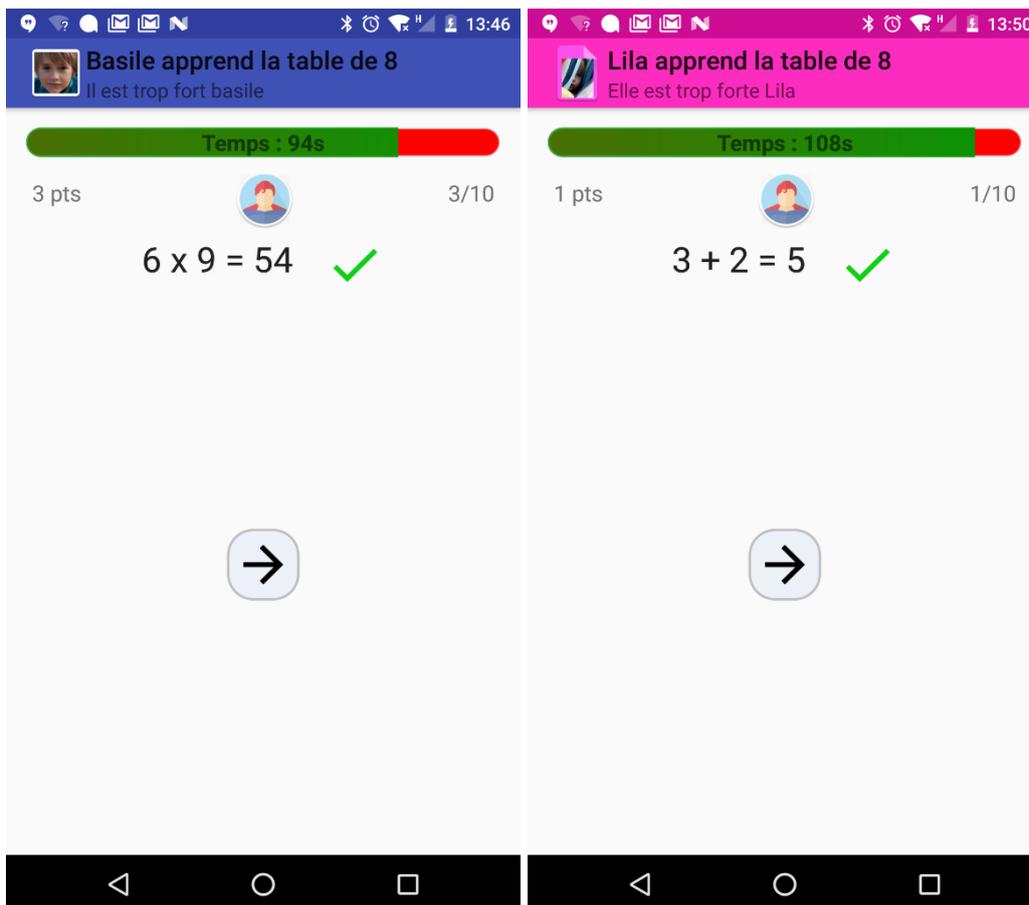
```
public class AssesmentOperation {  
  
    public static int caculate(int value,int factor){  
        return value+factor;  
    }  
}
```

And it's done.

In my main flavor, I just call that method like that:

```
multiplicatorFactor[i]=  
getMultiplicatorValue(position,multiplicationTable);  
value[i]= getValue(position, multiplicatorFactor[i],multiplicationTable);  
answer[i]=AssesmentOperation.caculate(value[i],multiplicatorFactor[i]);
```

And here comes the result:



Chapter 4: Flavors And Manifest

There is a specific use case I want to talk when playing with flavors.

Imagine you have made 2 flavors WhiteBands and CatBus in the dimension “brand” and your main flavor contains all your architecture code.

That way, setting your architecture in main, you don’t have to copy/paste project to reuse your architecture or create sdk that imprisoned your developers or worst a framework that not even created is already deprecated.

The flavor WhiteBands is the project where you code a sample application that show cases to all the others teams how to use the architecture and how to code most of the use cases.

The flavor CatBus is the final product. The objective is to be able to use your architecture, to reuse the “good pattern” code of WhiteBands and to be free to implement what is needed to be.

And in this configuration, you know that you’ll have several teams playing the game; dog, bird, canary, horse and so on. So they will fork you initial project (with main and WhiteBands) and create their own flavor DogBus and go in the build of their product on safe foundations.

Funny game, isn’t it?

To make the parallel with my tutorial, you can imagine basile is WhiteBands, lila is CatBus and you have your comprehension.

So in this scenario, one main problem that occurs is the initialization of some libraries in the onCreate of your MainApplication object. Because we do a lot in our Application object and we need to define it in main for our architecture. And as you know, if you let a Class in main, you cannot overwrite it in a specific flavor. That works only for resources, not for java code. So you are screwed.

So, how can I initialize my library? I cannot overwrite the code of MainApplication using flavor.

In fact it’s not the right question, the question is:

- how can I create MyLilaApplication object in the flavor lila
- that extends the MainApplication defined in the main flavor and
- explain to android that the Application object of the application is MyLilaApplication ?

If I can do that, I initialize my library in the onCreate of MyLilaApplication object. And such a solution will also work for Services and BroadcastReceivers defined in the main flavor.

So let’s do that.

8 In practice

Ok, so it’s simple until it won’t be.

I create my new MyLilaApplication in the Lila flavor that extends MainApplication defined in the main flavor.

I just overwrite onCreate and do a log, normally, you initialize elements here (like libraries).

```

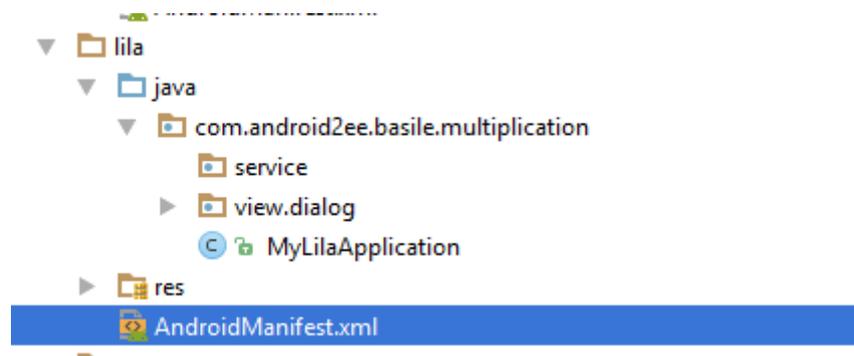
public class MyLilaApplication extends MainApplication {

    @Override
    public void onCreate() {
        super.onCreate();
        //I do something
        //like intializing a specific library
        Log.e("MyAppInitializer", "Second choices, a log is enough to prove
the concept: MyLilaApplication");
    }
}

```

Then the next step is explain to the system that Lila's flavor application object is implemented by MyLilaApplication.

We do that as usual in the manifest (the one of the flavor Lila):



So I update the manifest like this:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android2ee.basile.multiplication">

    <application
        android:allowBackup="true"

        android:name=".MyLilaApplication"

        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name=".view.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".view.AssessmentActivity"></activity>
    </application>

</manifest>

```

And I run my project. The Gradle build fails with this marvelous error:

Error:Execution failed for task ':app:processLilaAdditionDebugManifest'.

> Manifest merger failed : Attribute application@name
value=(com.android2ee.basile.multiplication.MyLilaApplication) from AndroidManifest.xml:6:9-38

is also present at AndroidManifest.xml:6:9-38
value=(com.android2ee.basile.multiplication.MainApplication).

Suggestion: add 'tools:replace="android:name"' to <application> element at
AndroidManifest.xml:6:5-21:19 to override.

Et voilà, shit happened.

So what is the problem? The problem comes from the merging resources process. As it's simple for the resources itself to be merge (straight overwrite strategy), the manifest is more complex to merge and it cannot be always done in a generic way. And here is the case where generic merging strategy fails.

Is it over? Are we screwed again ?

No, Google provides us tools to explain the merging strategy to adapt with our use case (and also they explain us what to do in the error's logs).

<https://developer.android.com/studio/build/manifest-merge.html>

The basic principle is that you can tune the merge of the different manifests. Basically when you have several manifests to merge they inherit from each other's, you just have to say, in the final merge, how to consider inherited attributes. You can:

- merge
- mergeOnlyAttributes
- remove
- removeAll
- replace
- strict (generate an error when merging conflict and stop)

The explanation page is really clear, just have a look and bookmark it :)

For us, we want to replace the inherited value of android:name by the new one :

```
<?xml version="1.0" encoding="utf-8" ?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android2ee.basile.multiplication"

    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"

        tools:replace="android:name"

        android:name=".MyLilaApplication"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".view.MainActivity">
```

```

        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER"
/>
        </intent-filter>
    </activity>
    <activity android:name=".view.AssessmentActivity"></activity>
</application>

</manifest>

```

Don't forget to add the name space of tools in your xml (the xmlns tag at the top).

And that's enough.

So if we remember the code:

```

public class MyLilaApplication extends MainApplication {

    @Override
    public void onCreate() {
        super.onCreate();
        //I do something
        //like intializing a specific library
        Log.e("MyAppInitializer","Second choices, a log is enough to prove
the concept: MyLilaApplication");
    }
}

```

We can run the application and obtain:

```
03-07 21:18:23.764 13926-13926/? E/MyAppInitializer: Second choices, a log is enough to prove the
concept: MainApplication
```

```
03-07 21:18:23.764 13926-13926/? E/MyAppInitializer: Second choices, a log is enough to prove the
concept: MyLilaApplication
```

So the log of the super class MainApplication (the one coded in the flavor main) is done first, because we call super.onCreate first (in MyLilaApplication) and the one of MyLilaApplication is done second, like the code expect.

So we do it, yes.

8.1 The getInstance question

Doing this, a question raised in my mind, I have a singleton pattern set on the MainApplication object. It should work fine, but am I sure?

So here is the pattern I definitely use (abuse) in my application's architecture?

```

public class MainApplication extends Application {
    /*****
    * Singleton
    *****/
    private static MainApplication instance;
    public static MainApplication ins(){
        return instance;
    }
}

```

```

/*****
 * Managing Life Cycle
 *****/
@Override
public void onCreate() {
    super.onCreate();
    instance=this; //Other code after
}

```

I use it a lot in my application for obtaining a context :

```
MainApplication.ins().getSharedPreferences().
```

But also for the ServiceManager, the ThreadManager...

Let's test a stuff, just let's make some logs

```

Log.e("Application_whoRU", "The instance of MainApplication.ins()=="+
    +MainApplication.ins().getClass().getSimpleName());
Log.e("Application_whoRU", "The instance of MainApplication.ins()=="+
    + MyLilaApplication.ins().getClass().getSimpleName());

```

And it returns:

```
03-07 21:28:05.821 E/Application_whoRU: The instance of MainApplication.ins()==MyLilaApplication
```

```
03-07 21:28:05.821 E/Application_whoRU: The instance of MainApplication.ins()==MyLilaApplication
```

Ok, it's cool. And watching the result, it's clear, we didn't have to worried, and it was obvious. But stupid question has to be answered, and when in doubt, just test. Having doubts is a good beginning.

So nothing to change, the singleton stays in MainApplication and flavors can use it safely.

Chapter 5: Understanding the life cycle.

Let's have an example to better understand a fundamental notion on gradle which is:

Don't forget about the build phases

A task has both configuration and actions. When using the <<, you are simply using a shortcut to define an action. Code defined in the configuration section of your task will get executed during the configuration phase of the build regardless of what task was targeted. [See Chapter 22, The Build Lifecycle](#) for more details about the build lifecycle.

https://docs.gradle.org/current/userguide/more_about_tasks.html

What did this sentence mean? Let's have an example.

9 The problem

Your goal is to enable Crashlytics when you are releasing a version but to disable it during development phase. So to do that you create a variable called enableCrashlytics and you use it, in your code, in your application object to enable/disable Crashlytics depending on that value.

So imagine in my build.gradle file I have such a code:

```
apply plugin: 'com.android.application'  
apply from: 'enableFabrics.gradle'
```

Where the enableFabrics.gradle file could be

Either that one

```
//Enable crashlytics  
task enableCrashlytics1 {  
    doLast {  
        //crashlytics (fabric)  
        android.defaultConfig.resValue "string", "enableCrashlytics", "true"  
    }  
}
```

Either that one:

```
//Enable crashlytics  
task enableCrashlytics2 {  
    //crashlytics (fabric)  
    android.defaultConfig.resValue "string", "enableCrashlytics", "true"  
}
```

And my gradle task for releasing could be that:

```
//This is the root task  
task generateWeeklyReport(dependsOn: ['enableCrashlytics*']) {  
    println 'Task generateWeeklyReport processing...'  
    //I do nothing, In am an entry point
```

```
//I use task dependency to launch the build
}
```

10 Comprehension

How gradle works? First Gradle will evaluate your project and build its gradle graph. It does that during the configuration phase. And to do that it runs all the code that are in configuration bloc.

What is a configuration bloc? It's a bloc that doesn't belong to an execution action. It means if your code is at the root of the task, not encapsulate in a doLast or doFirst blocs (there should be others), it will run whatever happens at configuration phase.

And when is configuration phase? Each time gradle needs to run a task it starts by the configuration phases, so each time you make a build/compile/assemble/cAT.. that code runs.

11 Solution

It means when you use the task enableCrashlytics1, its code is ran only at the execution time, when the task is called. So when you launch your release task. The rest of the time, the variable **enableCrashlytics** is set to its default value, false.

```
//Enable crashlytics
task enableCrashlytics1 {
    doLast {
        //crashlytics (fabric)
        android.defaultConfig.resValue "string", "enableCrashlytics", "true"
    }
}
```

But if you use enableCrashlytics2. The code is always ran at configuration phase, so the value of the variable **enableCrashlytics** is always set to true.

```
//Enable crashlytics
task enableCrashlytics2 {
    //This code is always executed at configuration phase
    //chrashlytics (fabric)
    android.defaultConfig.resValue "string", "enableCrashlytics", "true"
}
```

12 Conclusion

Here are the tips so:

- When you make a task, split your code between execution bloc (using doLast, doFirst) and your configuration bloc.
- Less you have code running at configuration phase, faster are your builds.
- Take the time to define which variable has to be defined and initialized depending on which tasks use it.

Chapter 6: Setting code coverage on Android with Jacoco

You also want to have your code coverage on Android, but it doesn't work, there is a lot a tutorial explaining how to do, github project, gradle plugin... but all this resources (most of them are deprecated but you don't know it) are just messing our mind and we finish by not being sure if Android build system is able or not to generate code coverage.

But it should be simple, and in fact, it is. So instead of giving you the receipt, I prefer to give you the understanding. Because we want and need that:

app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.kameleon.service.core.cross.exception.devreporter	100%	79%	3	15	0	37	0	8	0	1		
org.kameleon.service.core.cross.manager.analytics	100%	75%	1	7	0	35	0	5	0	1		
org.kameleon.service.mycms.garage.view.holders	100%	50%	1	6	0	20	0	5	0	2		
org.kameleon.service.core.com.common.interceptors	100%	50%	2	8	0	19	0	6	0	3		
org.kameleon.service.core.view.toolbar.holder	100%	100%	0	6	0	16	0	5	0	2		
org.kameleon.service.mycms.garage.view.adapters	100%	n/a	0	4	0	11	0	4	0	1		
org.kameleon.service.mycms.crossfeature.service	100%	100%	0	3	0	9	0	2	0	1		
org.kameleon.service.mycms.crossfeature.com.interceptors	100%	n/a	0	2	0	9	0	2	0	1		
org.kameleon.service.core.test.view	100%	n/a	0	6	0	11	0	6	0	1		
org.kameleon.service.core.cross.exception.userreporter	100%	n/a	0	2	0	5	0	2	0	1		
org.kameleon.service.core.com.manager	100%	n/a	0	1	0	2	0	1	0	1		
org.kameleon.service.core.view.generic.indeterminatedecorator.drawable	100%	n/a	0	1	0	1	0	1	0	1		
org.kameleon.service.core.view.toolbar.adapter	96%	75%	1	6	0	11	0	4	0	1		
org.kameleon.service.core.cross.exception	96%	73%	9	46	4	107	1	31	0	3		
org.kameleon.service.mycms.crossfeature.com.wrapper	95%	71%	5	9	3	28	1	2	0	1		
org.kameleon.service.core.com.common	94%	79%	4	16	2	35	1	9	0	1		
org.kameleon.service.core.view.generic.indeterminatedecorator	93%	79%	5	25	4	76	0	13	0	2		
org.kameleon.service.core.cross.manager.connectivity	92%	62%	20	50	12	101	0	19	0	3		
org.kameleon.service.core.view.view.legacy	91%	50%	10	61	9	133	6	57	0	6		
org.kameleon.service.core.service.executor	89%	88%	1	14	7	35	0	10	0	4		
org.kameleon.service.core.view	88%	63%	28	77	33	219	9	51	0	6		
org.kameleon.service.core.view.generic.gmap	87%	58%	10	18	2	31	0	5	0	1		

And when you have understood it's easy to set even with a bunch of flavor dimensions.

So the goal is to have a final report on your tests coverage for all your tests. Let's do that.

13 Enabling Jacoco on your project

By default, you have Jacoco in your Android build system, you just have to enable it and you'll have your first report each time you run tests.

Do to that, you have to enable it in 2 different places in your build.gradle.

13.1 Enable code coverage for instrumented tests

You activate it for the build you want (debug or release):

```
buildTypes {
    release {
        //your debug part
        //add tests coverage using Jacoco
        testCoverageEnabled false
    }
    debug {
        //your debug part
        //add tests coverage using Jacoco
        testCoverageEnabled true
    }
}
```

Doing that enable your tests reports for all you instrumented test for your Debug build type. As I only need it for Debug, I just set it on the Debug build type.

13.2 Enable tests coverage for UnitTests

To enable your code coverage for unit tests add this gradle hook (in your build.gradle or in a separate file called by your build.gradle)

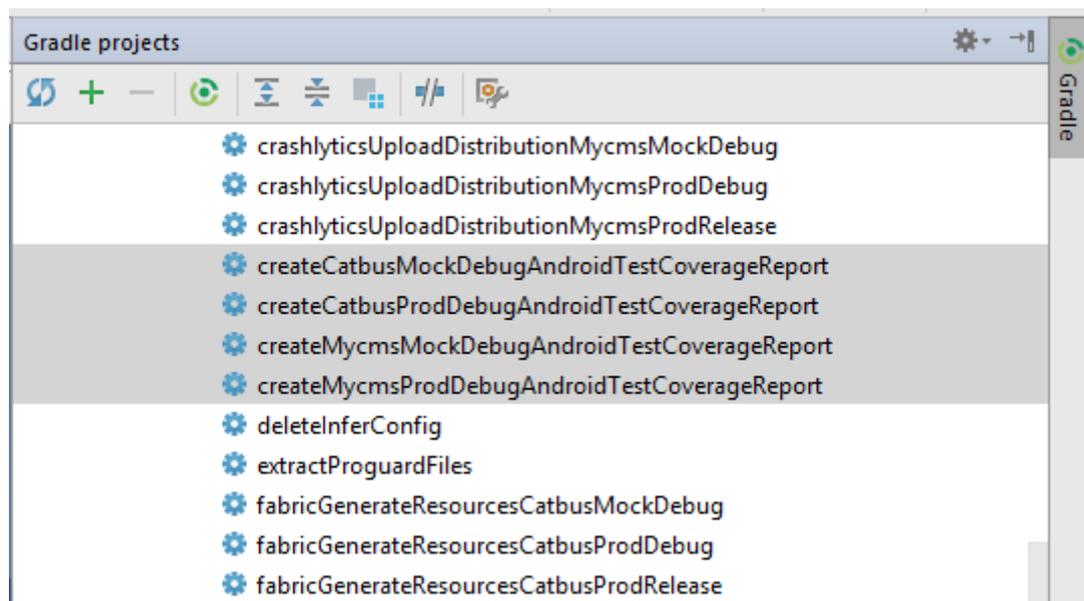
```
//Enable the coverage report for unit test
android.testOptions {
    unitTests.all {
        jacoco {
            includeNoLocationClasses = true
        }
    }
}
```

You did it, tests coverage is enable on the project.

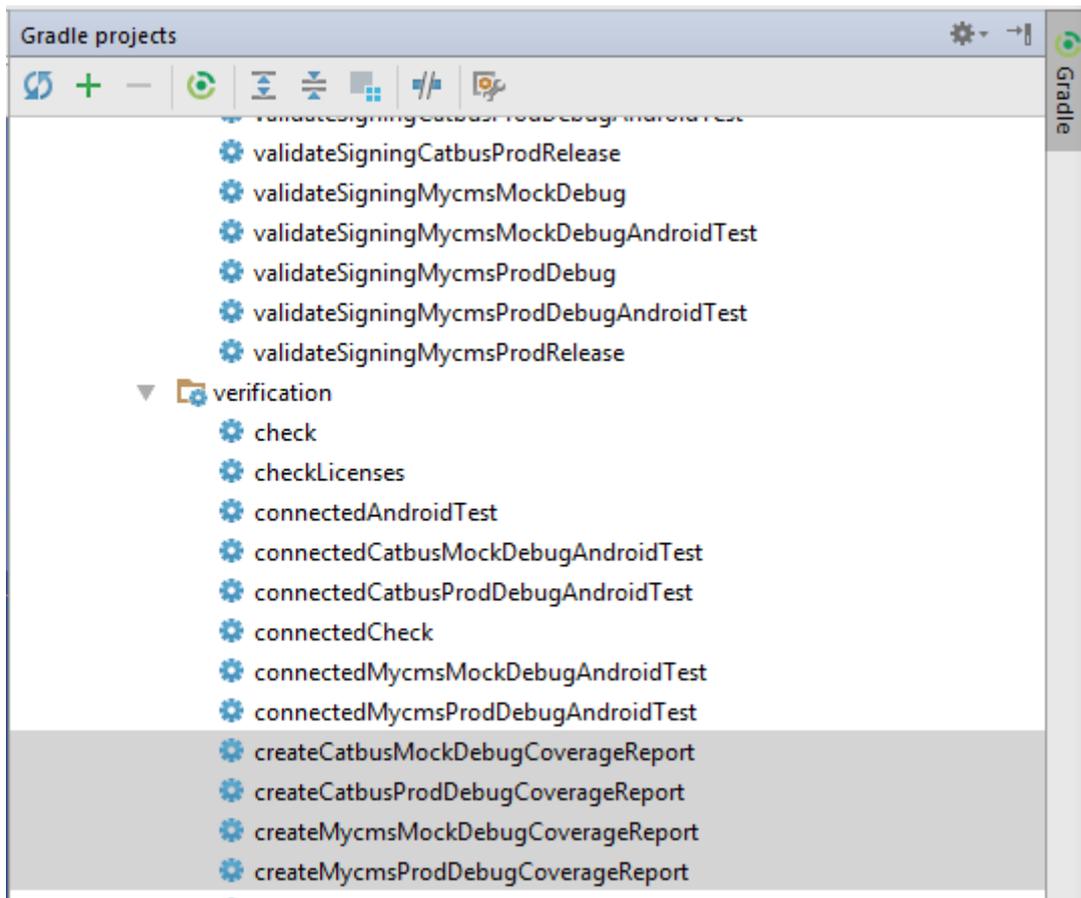
13.3 What happened

Doing so, you have created in your project several gradle tasks that are here to generate your .ec files (raw data of coverage report) for your instrumented tests.

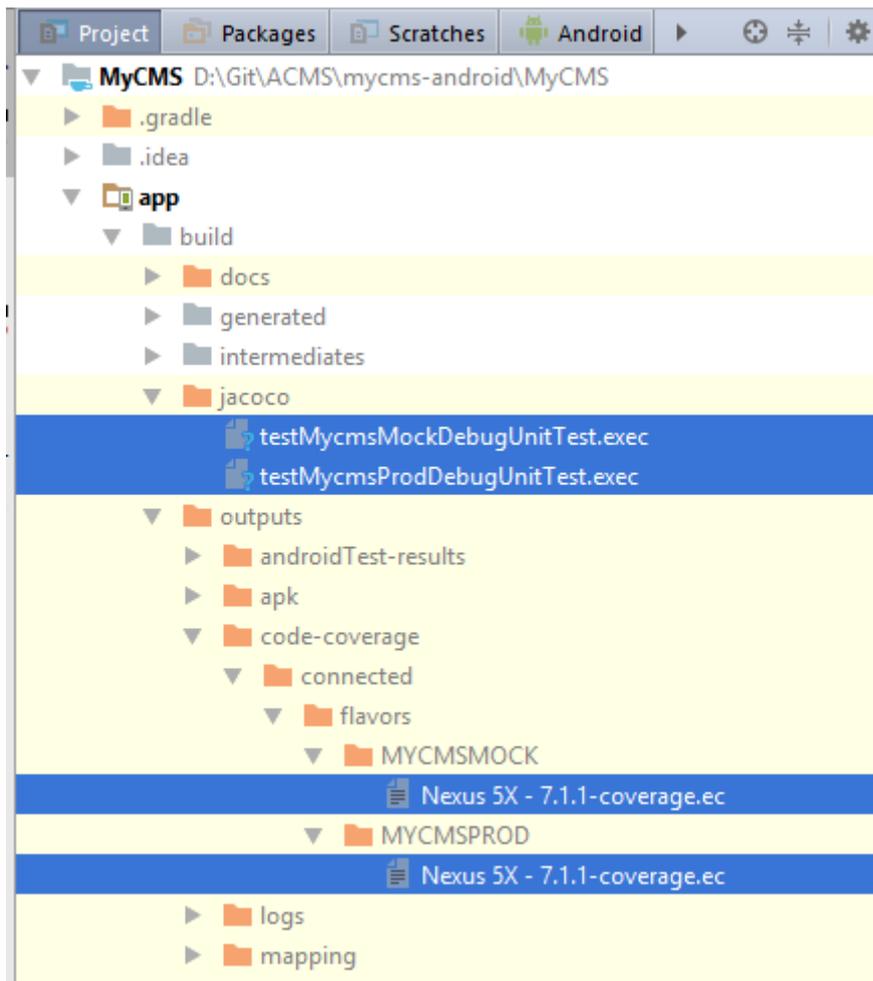
You can find them in the category other.



And the same tasks for your unit tests reports (category verification)



Now, each time you run your tests, those tasks are executed and jacoco will generate the raw file of the coverage report for this set of tests in a specific format. Exec for unit tests and ec for instrumented tests, you can find them in your build folder.



14 Defining and creating your report

Then let's create a specific gradle task, that will generate the report, this way, you can call it from AndroidStudio or directly with the gradle console.

The algorithm of this task is:

- Run your tests and generate your exec and ec files
- Define which files you want to exclude from the analysis (auto generated code, code from library, some of your classes that are tests classes, or pojo)
- Define where is the code to analyse; you need to give the location of the .class files (in build/intermediates) and the package they belong in your source code (for exemple src/main/java)
- Define which in which files are your raw coverage data generated by Jacoco (the .ec and .exec files)

And that's all. Crazy, I hunt this information in the 2 first google pages results of "jacoco android gradle" or such a request.

Now we know what to do, let's do it:

```

apply plugin: 'jacoco'
//Your task is task of type Jacoco
//and you have to run all your tests task and you create***CodeCoverage one
//those tasks are generated when you enable Jacoco (first chapter)
task jacocoTestReport(type: JacocoReport, dependsOn: [
    //if you have already run those tasks in your build, just comment them
    'testMycmsMockDebugUnitTest',
    'testMycmsProdDebugUnitTest',
    'createMycmsMockDebugCoverageReport',
    'createMycmsProdDebugCoverageReport',
]) {
    //Define which type of report you want to generate
    reports {
        xml.enabled = true
        html.enabled = true
    }
    //define which classes to exclude
    def fileFilter = [
        '**/R.class',
        '**/R$.class',
        '**/BuildConfig.*',
        '**/Manifest.*',
        '**/*$ViewInjector.*',
        '**/*$ViewBinder.*',
        '**/*$Lambda$*.*/', // Jacoco can not handle several "$" in class name.
        '**/*Module.*', // Modules for Dagger.
        '**/*Dagger.*', // Dagger auto-generated code.
        '**/*MembersInjector.*', // Dagger auto-generated code.
        '**/*_Provide*Factory.*',
        '**/*_Factory.*', //Dagger auto-generated code
        '**/*$*$.*', // Anonymous classes generated by kotlin
    //add libraries
        'android/**/*.*',
        'com/**/*.*',
        'uk/**/*.*',
        'io/**/*.*',
    //remove what we don't test
        'androidTest/**/*.*',
        'test/**/*.*',
        '**/injector/**/*.*',
        '**/model/**/*.*',
        '**/mock/**/*.*',
        '**/event/**/*.*',
        '**/*_ViewBinding**',
        '**/*EventType.*',
        '**/*Mocked'
    ]

    //Define your source and your classes: we want to test the production code
    def debugTree = fileTree(dir: "${buildDir}/intermediates/classes/mycmsProd/debug", excludes: fileFilter)
    def mainSrc = files(["src/main/java", "src/mycms/java"])
    //Explain to Jacoco where is your source code
    sourceDirectories = files([mainSrc])
    //Explain to Jacoco where are you .class file
    classDirectories = files([debugTree])
    //As you want to gather all your tests reports, add the ec and exec you want to be took into
    //account when generating the report

```

```
executionData = fileTree(dir: "$buildDir", includes: [  
    "jacoco/testMycmsMockDebugUnitTest.exec",  
    "jacoco/testMycmsProdDebugUnitTest.exec",  
    "outputs/code-coverage/connected/flavors/**/*coverage.ec"  
])  
}
```

15 References

https://medium.com/@rafael_toledo/setting-up-an-unified-coverage-report-in-android-with-jacoco-robolectric-and-espresso-ffe239aaf3fa

and

<https://medium.com/contentsquare-engineering-blog/make-or-break-with-gradle-dac2e858868d>

Chapter 7: Build Organization

On this chapter, I want to focus on what I want, not what I can do. So let's make our wishes list for our build file.

Concerning organization of the builds, I want to:

- extract my sensitive information in a specific file excluded from Git (security)
- split my build.gradle file in more specific build files (uploading, ...)
- Define specific resources values that can be handled by my java code, my resources and the manifest.

Concerning application's builds, I want to:

- add a flavor mock and prod for tests environment
 - build my apk in debug mode with a mock flavor for running instrumentation tests,
 - build my apk in debug mode with a production flavor to test it on the team's devices,
- build my apk in release mode, signed, proguarded, minified and resources shrank
- be able to run all those apk on the same device
- increment my versionCode automatically
- tune my lint options
- manage dex problems using an already multi-dex library in the project

Concerning flavors management, I want to:

- Define several flavors dimensions: Mocked (with prod and mock), TeamSpecific (Ux, AlphaTests, ...), Brands (Ambre, CatBus, Cars...), others
- and do those builds for each of those flavors
- Share java code between dimensions (a good way to define default architecture behaviors easily over writable)
- Do the same with resources

Concerning tests and validation, I want to:

- be able to run my tests and have reporting (lint reports, tests coverage, tests reports) in specific directories
- be able to drop it on Jenkins
- be able to include spoons to run parallel instrumentations tests with screen captures on the devices/emulators
- be able to deploy a snap-shot of the project to my team with docs for check points (it means create a directory with the apk, the release note, the lint and tests reports for team validation...) and zip it

Concerning deployment, I want to:

- be able to generate JavaDoc,
- be able to deploy it on a Nexus (maven central private repository) with its sources, javadocs

Concerning Stores (GooglePlay), I want to:

- split apk depending on screen densities for a smooth delivery on googleplay

- Prepare a folder with all the needed files for the GooglePlay deployment

We won't explain all those dots, but you should be able to set them all when you have finished the reading of those chapters.

When writing this article I rely a lot on:

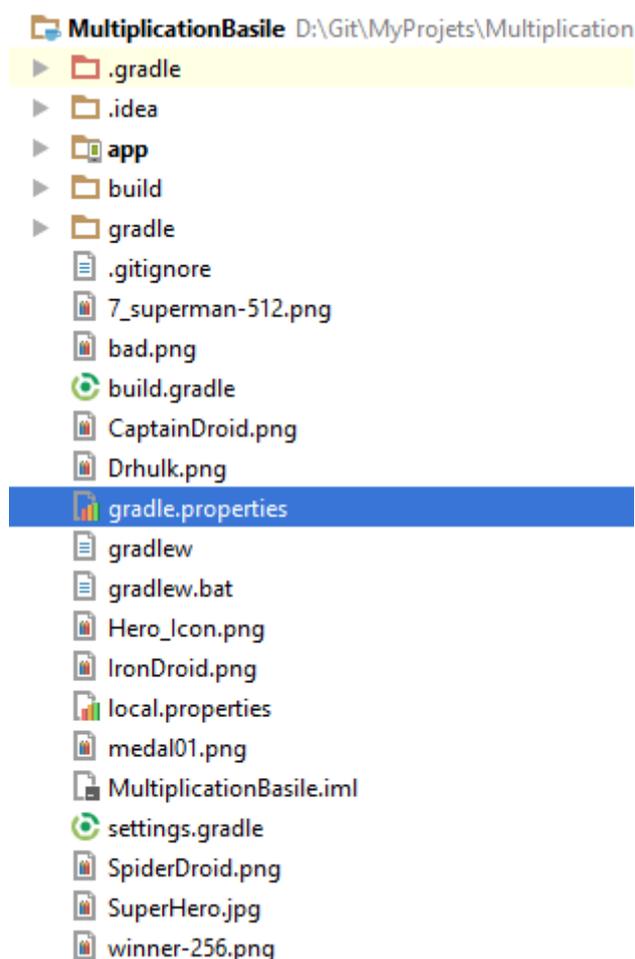
<https://developer.android.com/studio/build/gradle-tips.html#configure-project-wide-properties>

16 Builds Organization

16.1 General notions on *.properties files

We need to be able to consider our build.gradle as a function that makes stuff and being able to feed this function with parameters. So here comes the properties files.

The default properties file for your project is gradle.properties at the project root.



In this file, you will be able to define attributes, variables that will be visible in your build.gradle files.

The default content of this file looks like that:

```
# Project-wide Gradle settings.

# IDE (e.g. Android Studio) users:
# Gradle settings configured through the IDE *will override*
# any settings specified in this file.

# For more details on how to configure your build environment visit
```

```
# http://www.gradle.org/docs/current/userguide/build_environment.html

# Specifies the JVM arguments used for the daemon process.
# The setting is particularly useful for tweaking memory settings.
org.gradle.jvmargs=-Xmx1536m

# When configured, Gradle will run in incubating parallel mode.
# This option should only be used with decoupled projects. More details, visit
#
http://www.gradle.org/docs/current/userguide/multi_project_builds.html#sec:decouple
d_projects
# org.gradle.parallel=true
```

16.2 Defining your own properties files

When you want to create a new properties file for our project, just create it and add it to the build.gradle where you want to use it, like that:

```
// Creates a variable called keystorePropertiesFile, and initializes it to the
// keystore.properties file.
def keystorePropertiesFile = rootProject.file("keystore.properties")

// Initializes a new Properties() object called keystoreProperties.
def keystoreProperties = new Properties()

// Loads the keystore.properties file into the keystoreProperties object.
keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
```

(<https://developer.android.com/studio/build/gradle-tips.html#configure-project-wide-properties>)

To use it in your file, just use this syntax to access the values

```
keystoreProperties['keyAlias']
```

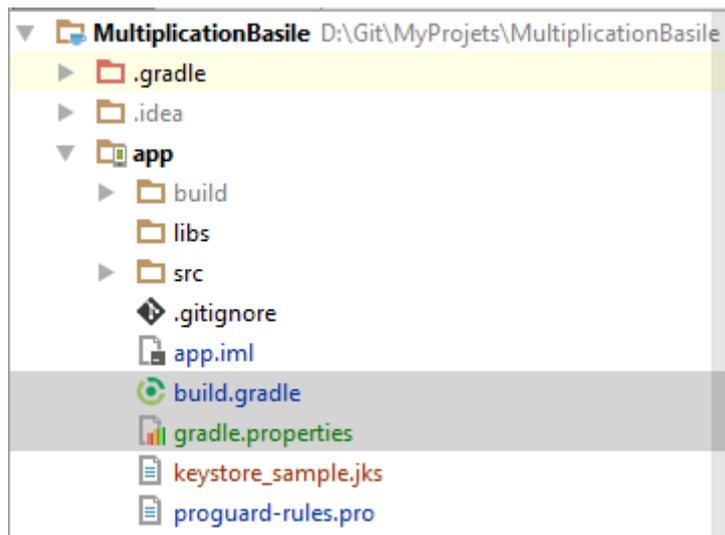
16.3 Extracting password from gradle.properties file (Default)

You should use properties if:

- You have sensitive information (like id/password) in your build.gradle, you need to extract it
- You have configuration information (server urls, server deployment, specific directories)
- You have project information (dev name and role)

You should extract those information in specific properties files. You can exclude some files from you versioning sources control (Git) to protect sensitive information or specific configurations.

So, in our project, let's create some properties in the default gradle.properties file:



In the file gradle.properties, you just define your variables:

```
#Define your name
#-----
POM_DEVELOPER_ID=MSE_A2EE
POM_DEVELOPER_NAME=Seguy Mathias
POM_DEVELOPER_MAIL=mathias.seguy@android2ee.com
POM_DEVELOPER_ROLE=Android Referent Expert
#Signing configurations
STORE_FILE=keystore_sample.jks
STORE_PASSWORD=android
KEY_ALIAS=android
KEY_PASSWORD=android
```

And you use them in your build.gradle:

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.android2ee.basile.multiplication"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }

    signingConfigs {
        release {
            storeFile file(STORE_FILE)
            storePassword STORE_PASSWORD
            keyAlias KEY_ALIAS
            keyPassword KEY_PASSWORD
        }
    }

    buildTypes {
        release {
            signingConfig signingConfigs.release
            minifyEnabled false
            shrinkResources false
            useProguard false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}
```

```

    debug {
        applicationIdSuffix '.debug'
        versionNameSuffix '.debug'
    }
}

```

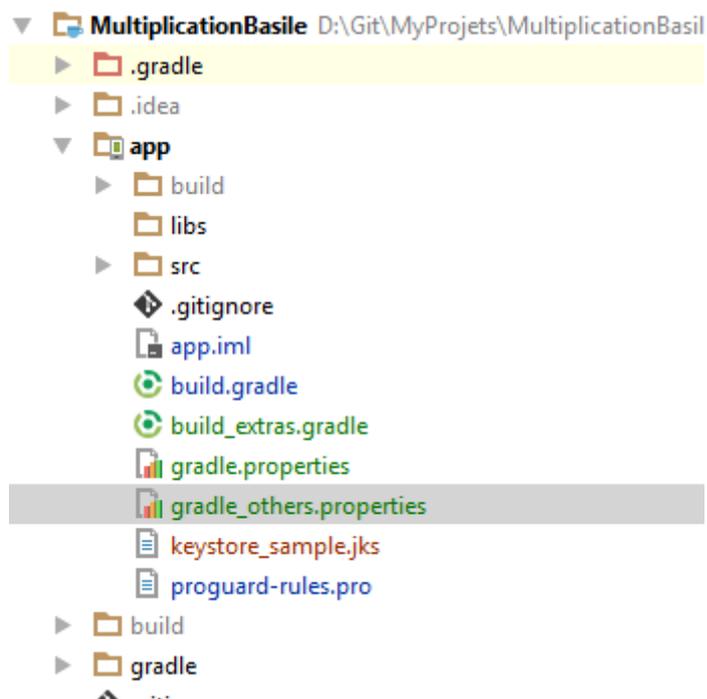
You have externalized confidential information from the build file. If you want to keep them away from your version control, you can.

16.4 Extracting password from my gradle_others.properties

So let's say, we have extract the information but not in gradle.properties but in a file called gradle_others.properties. How does it works?

It's almost the same, but here, you have to load the file by yourself, like this:

The file ./app/gradles_others.properties



Contains the following variables definition:

```

#Signing configurations
STORE_FILE=Not set yet
STORE_PASSWORD=android
KEY_ALIAS=android
KEY_PASSWORD=android

#URL REPO FAKE DATA
UPLOAD_REPO=http://notset
UPLOAD_USER=JohnDOE
UPLOAD_PASS=0000

```

So in the build.gradle file of my app module, I just add the line of code to load this properties. I do that in the android block, that way, it runs when the project is defined by gradle (for any tasks).

Note that I need to define my variables before trying to use them.

```

apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"

    /*****
    * Defining and Loading Properties
    *****/
    //Define your own properties
    project.ext {
        uploadRepo = 'not_set'
        uploadUser = 'not_set'
        uploadPass = 'not_set'
    }

    //Define a Properties object (you'll read)
    def Properties props = new Properties()
    //find the file defining those properties
    def propFile = project.file('gradle_others.properties')
    //You can also use rootProject
    //def propFileNotUsed = rootProject.file('./app/gradle_others.properties')

    if (propFile.canRead()){
        //load the properties in your Properties object
        props.load(new FileInputStream(propFile))

        //You can define specific variables of the build directly
        if (props!=null && props.containsKey('STORE_FILE')) {
            //You can directly set variables of the project
            //here defining the debug signing config
            android.signingConfigs.debug.storeFile = file(props['STORE_FILE'])
        } else {
            println 'gradle_others.properties found but entry missing'
            android.buildTypes.debug.signingConfig = null
        }

        //Or you can set your own variables
        if (props!=null && props.containsKey('UPLOAD_REPO')){
            uploadRepo=props['UPLOAD_REPO']
            println 'gradle_others.properties found '+uploadRepo
        }else{
            println 'gradle_others.properties UPLOAD_REPO entry is missing'
        }
    } else {
        println 'gradle_others.properties not found'
        android.buildTypes.debug.signingConfig = null
    }

    /*****
    * Signing
    *****/
    defaultConfig {...}

    /*****
    * Signing
    *****/
    signingConfigs {
        release {...}
        debug {...}
    }
}

```

Wait, what happens here? Ok, let's do it step by step.

16.4.1 Define your variables

This bloc declare uploadRepo, uploadUser and uploadPass as variables of the project.

```
//Define your own properties
project.ext {
    uploadRepo = 'not_set'
    uploadUser = 'not_set'
    uploadPass = 'not_set'
}
```

16.4.2 Use your variable

They can be used anywhere after that, in your script or as variable of the build:

```
task printRepoVar() {
    println uploadRepo
}
```

Or

```
signingConfigs {
    debug {
        storeFile file(uploadRepo)
        storePassword STORE_PASSWORD
        keyAlias KEY_ALIAS
        keyPassword KEY_PASSWORD
    }
}
```

16.4.3 Load your properties file

The next step is to find the file and load it into a Properties object:

```
//Define a Properties object (you'll read)
def Properties props = new Properties()
//find the file defining those properties
def propFile = project.file('gradle_others.properties')
//You can also use rootProject
//def propFileNotUsed = rootProject.file('./app/gradle_others.properties')

if (propFile.canRead()){
    //load the properties in your Properties object
    props.load(new FileInputStream(propFile))
}
```

16.4.4 Initialize your variables

Then you use your properties object to set your variables:

Those of the build script itself:

```
if (props!=null
&& props.containsKey('STORE_FILE')
&& props.containsKey('STORE_PASSWORD')
&& props.containsKey('KEY_ALIAS')
&& props.containsKey('KEY_PASSWORD')) {
    //You can directly set variables of the project
    //here defining the debug signing config
    android.signingConfigs.debug.storeFile = file(props['STORE_FILE'])
    android.signingConfigs.debug.storePassword = props['STORE_PASSWORD']
    android.signingConfigs.debug.keyAlias = props['KEY_ALIAS']
    android.signingConfigs.debug.keyPassword = props['KEY_PASSWORD']
} else {
    println 'gradle_others.properties found but some entries are missing'
    android.buildTypes.debug.signingConfig = null
}
```

Or your own:

```
//Or you can set your own variables
if (props!=null && props.containsKey('UPLOAD_REPO')){
    uploadRepo=props['UPLOAD_REPO']
    println 'gradle_others.properties found '+uploadRepo
}else{
    println 'gradle_others.properties found but UPLOAD_REPO entry is missing'
}
```

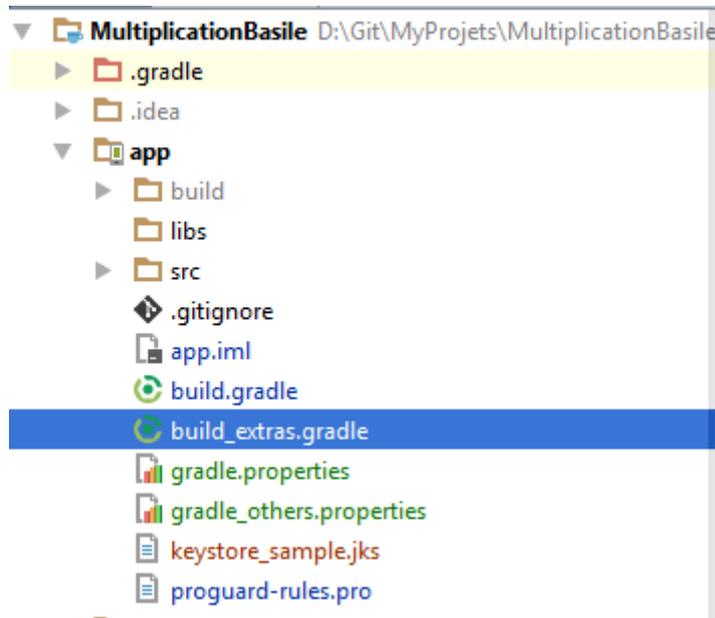
16.5 Splitting the build.gradle file into several files to gain in readability

More we do in our build.gradle, more the file goes long, complex and finally achieve to be unreadable.

To avoid this treat, we are going to split our build.gradle in several different files. Each of this file will be responsible of a specific part of the whole build.

For example, does the code to load properties needs to be in build.gradle? No, not really, we will gain in readability by extracting it.

Create your new file, here, I called it build_extras.gradle:



Then I extract the code from build.gradle and copy paste it in the build_extras.gradle files. This code defines the variables of the previous example and set them using the properties file.

My build_extras.gradle file looks like:

```
println 'Hello fomr gradle_extras'
/*****
 * Loading Variables
 *****/

//Define your own properties
project.ext {
    uploadRepo = 'not_set'
    uploadUser = 'not_set'
    uploadPass = 'not_set'
}
```

```
//Define a Properties object (you'll read)
def Properties props = new Properties()
//find the file defining those properties
def propFile = project.file('gradle_others.properties')
. . . . . }
```

Then, I just have to apply it in my build.gradle, in my android bloc

build.gradle:

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig { . . . }

    /*****
     * Signing
     *****/
    signingConfigs {
        release { . . . }
        debug { . . . }
    }

    /*****
     * Defining and Loading Properties
     *****/
    apply from: 'build_extras.gradle'
```

And that's it. This is the way we gonna avoid 10 thousands lines build.gradle file.

And that's important.

16.6 Define variables for Java code and Manifest

So we have defined variables for our gradle's build. What about define them, at this level, for some critic information in our code? Or our resources? Our Manifest?

We can define variables in two types of object:

- BuildConfig
- ResValues which are your resources

If you define variables in those classes, they'll appear as attributes for the classes:

- **resValue "string", "hidden_string", "I love you my sweety"**
Define in our resources, a string, with name hidden_string and value I love you sweety

And

- **buildConfigField("boolean", "isallowed", "true")**
Define in our BuildConfig object a Boolean called isallowed with a default value to true. You can omit the parenthesis.

You can define a default value for all the variants and then overwrite it in your flavors or buildType blocs. It's a good practice to always define default values.

There is limitation on the type of primitive you can use. It works well with String, Boolean and Colors. I didn't find the way to define boolean or int in ResValues. I neither try with all the primitives' types.

Let's have a look at the code.

```

//define default value for your attributes
defaultConfig {
    buildConfigField("boolean", "isallowed", "true")
    buildConfigField("String", "isStringallowed", "\"quarante
trois\"")
    buildConfigField("int", "intAllowed", "3")
    resValue "string", "hidden_string", "I love you my sweety"
    resValue "color", "color_var", "#FF00ff"
    resValue "bool", "isBoolAllowed", "true"
}

/*****
* Build Type
*****/

buildTypes {
    release {
        signingConfig signingConfigs.release
        minifyEnabled false
        shrinkResources false
        useProguard false
        buildConfigField("boolean", "isallowed", "false")
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
    }
    debug {
        applicationIdSuffix '.debug'
        versionNameSuffix '.debug'
        buildConfigField("boolean", "isallowed", "true")
        resValue "string", "hidden_string", "I love you debug"
    }
}

/*****
* Managing flavors
*****/

//Give a name to your dimension
flavorDimensions "enfants", "operator"

//define your flavors (as one flavor has a dimension they must all have one)
productFlavors {
    basile {
        dimension "enfants"
        //resValue "boolean","basile_dimension","true" -> not allowed
        //resValue "int", "int_allowed", "1" -> not allowed
        buildConfigField("boolean", "isallowed", "true")
        resValue "string", "hidden_string", "I love you my basilou"
        resValue "color", "int_allowed", "#FF00ff"
    }
    lila {
        dimension "enfants"
        applicationId "com.android2ee.lila.multiplication"
        buildConfigField("boolean", "isallowed", "true")
        resValue "string", "hidden_string", "I love you my Lila"
        resValue "color", "int_allowed", "#FF00ff"
    }
    multiplication {
        dimension "operator"
    }
    addition {
        dimension "operator"
    }
}
}

```

And I can use them like that:

```

public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Log.e("MyAppInitializer", "Gradle Variable resValues.hidden_string
        =" + R.string.hidden_string);

        Log.e("MyAppInitializer", "Gradle Variable
        resValues.isBoolAllowed=" + R.bool.isBoolAllowed);

        Log.e("MyAppInitializer", "Gradle Variable
        resValues.color_var=" + R.color.color_var);

        Log.e("MyAppInitializer", "Gradle Variable
        BuildConfig.isallowed=" + BuildConfig.isallowed);

        Log.e("MyAppInitializer", "Gradle Variable
        BuildConfig.isStringallowed=" + BuildConfig.isStringallowed);

        Log.e("MyAppInitializer", "Gradle Variable
        BuildConfig.intAllowed=" + BuildConfig.intAllowed);
    }
}

```

Returns :

E/MyAppInitializer: Gradle Variable resValues.hidden_string =2131099696

E/MyAppInitializer: Gradle Variable resValues.isBoolAllowed=2131296261

E/MyAppInitializer: Gradle Variable resValues.color_var=2131361814

E/MyAppInitializer: Gradle Variable BuildConfig.isallowed=true

E/MyAppInitializer: Gradle Variable BuildConfig.isStringallowed=quarante trois

E/MyAppInitializer: Gradle Variable BuildConfig.intAllowed=3

It also works for the Manifest when you have defined them as ResValues (BuildConfig is not accessible):

```

-->
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="@string/google_maps_key" />
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version"/>
</application>

```

16.7 Custom my gradle memory to run fast

If you have a good engine under the keyboard, you can tune gradle to be more efficient. You have to add in the gradle.properties file the following lines:

```

#Enable daemon
org.gradle.daemon=true

# Specifies the JVM arguments used for the daemon process.
# The setting is particularly useful for tweaking memory settings.
# Try and findout the best heap size for your project build.
org.gradle.jvmargs=-Xmx2048m -XX:MaxPermSize=512m -
XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8

# When configured, Gradle will run in incubating parallel mode.
# This option should only be used with decoupled projects. More details,
visit
#
http://www.gradle.org/docs/current/userguide/multi_project_builds.html#sec:
decoupled_projects
# Modularise your project and enable parallel build
org.gradle.parallel=true

# Enable configure on demand.
# avoid building part of the project when it's not necessary
org.gradle.configureondemand=true

```

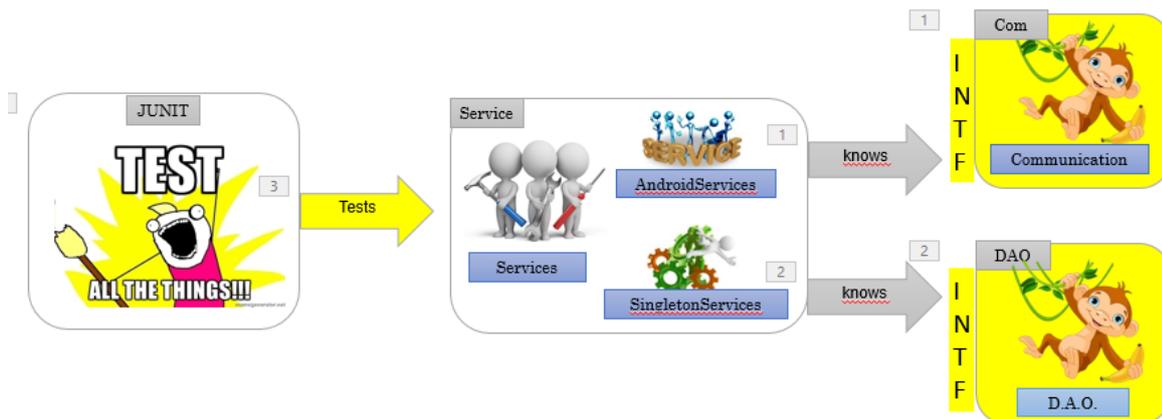
You can increase you're your heap for gradle, just change the org.gradle.jvmargs to 3072 for example or more if you are lucky.

17 Application's builds

17.1 Mock and Production flavors for tests environments

When I build an application, one of my concern is testing. So I want to be able to test my application. It means if I want to test a piece of my application, I need to fix inputs of the element to tests its outputs.

For example, if I want to test one of my business service, I will do something like that:



If we look at that diagram, we clearly see that for testing the service, I need to have mocked communication and mock DAO, because it's my inputs. And the service itself is the real service. As I want to test all elements of my application, that means we have in the test apk all the real elements of the application AND all the needed mocked elements (View, Services, Com, DAO...).

How can I build such an application? Do I need to embed in my production application all those mocked classes just for testing? Of course not.

So I need to have a specific flavor, let's call it `test_env` (for test environment context) with two values. The first one mocked which contains all my mocked elements and all my application's elements. The second one, which is production, contains only the real classes without the mocked elements.

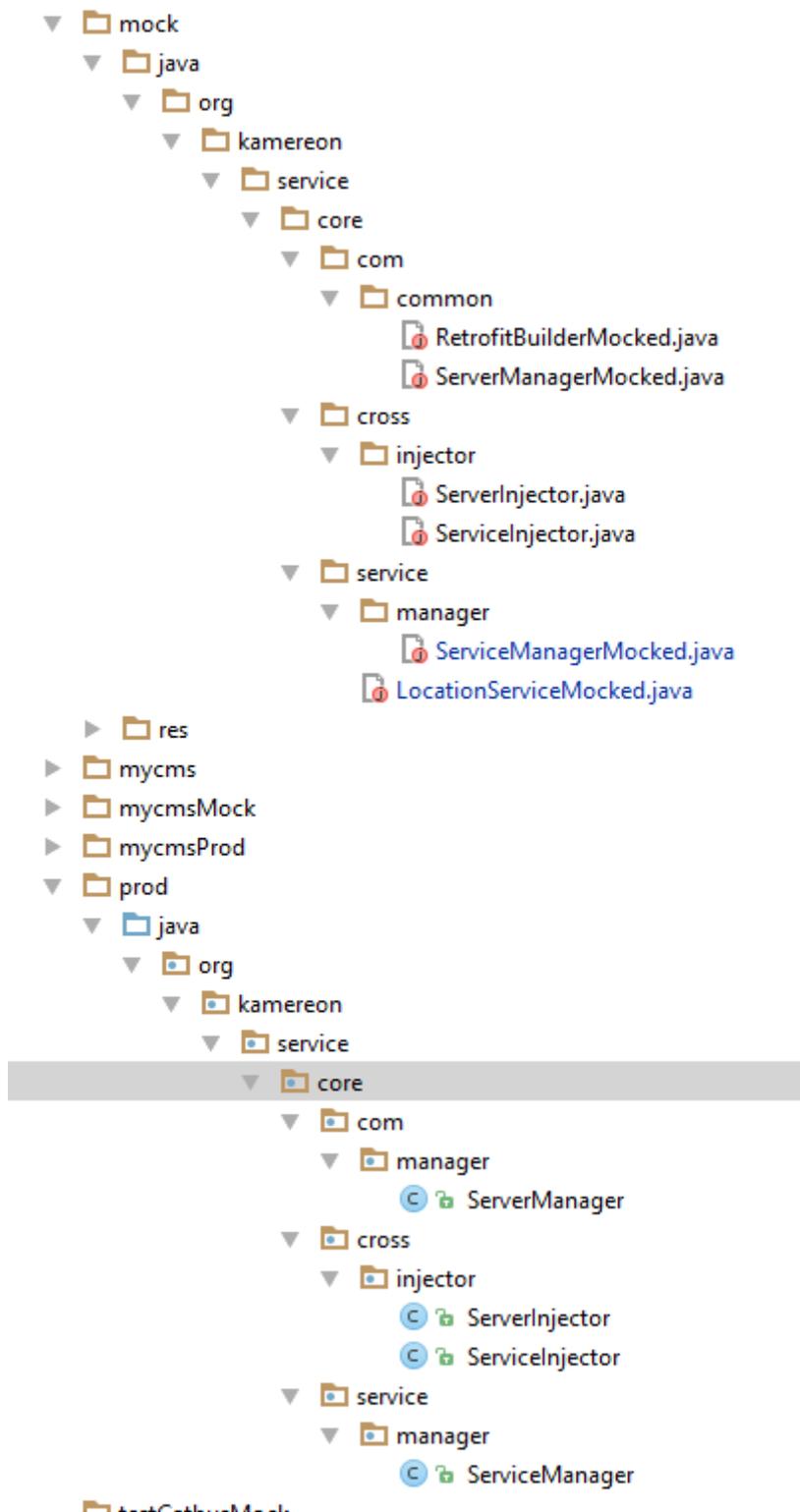
Let's do that:

```
/******  
 *   Managing flavors  
******/  
  
productFlavors {  
    prod {  
        dimension "test_env"  
        versionNameSuffix "-prod"  
        applicationIdSuffix ".prod"  
    }  
    mock {  
        dimension "test_env"  
        versionNameSuffix "-mock"  
        testProguardFile('proguard-rules-test.pro')  
        applicationIdSuffix ".mock"  
    }  
}
```

As I want to install those both applications on the same device (I want to avoid package conflicts and be able to run each application variant), I have changed the application Id. For me to have a good description in my device of the applications, I have changed its version name.

In my application I have centralized the managements of my services, dao, com through managers. For example, I have the `ServiceManager` and in my application, I access services through it, like `ServiceManager.getDataService()`. The same for `DaoManager`, `ComManager`... If I have implemented the `ServiceManager` in the flavors mock and production the following way. In mock the services return will be mock services, in production, returned services will be the real ones. If I use interfaces for my services, I can use them in my application and use the `**Manager` to retrieve the real class depending on this flavor context. It's based on the interface-factory pattern.

So I just have to inject the right `ServiceManager` (Mock or Prod) depending on the flavor. To do that, I prefer using the injector pattern:



17.1.1 Debug mode with mock flavor for the instrumentation tests

17.1.2 Debug mode with prod flavor for the team's devices (to tests)

17.2 Deliver the project for weekly delivery to the team and stakeholders

The goal here is to deliver your work to your stakeholders automatically. To do that we will create a specific gradle task to make the job and use Jenkins to run this task weekly.

But what do I want to do for this release?

I want to deliver the real product in its most stable state. This means, we want to deliver the master branch in its production flavor.

I also want to deliver the real product in its current state. This means, we want to deliver the dev branch in its production flavor.

By the way, it happens that we want to test the application, using a tests cases plan. This plan needs fixed input to check if the outputs are ok. In a way, instead of running automatically the tests using instrumentation tests, a human will do that.

So I also want to deliver the product in its most stable state and in its testable state (meaning I want to use the mock flavor that insure the inputs are always the same).This means, we want to deliver the master branch in its mock flavor.

So I also want to deliver the current product in its testable state (meaning I want to use the mock flavor that insure the inputs are always the same).This means, we want to deliver the dev branch in its mock flavor.

To sum up, we want:

For the master branch: prod_debug, mock_debug

For the dev branch: prod_debug, mock_debug

Then we also want to add reports to this delivery:

- Tests reports
- Lint reports
- Analysis tools reports
- Splash screens

As the project evolves, we will also include the CatBus project in the delivery as a demonstration of the brand customization of the application.

And as we want you can install all those apk on the same device (your devices) they will all have a different ApplicationId, Application name and a launcher icon (if we don't forget the icon).

And this wil belong to the next chapter.

Chapter IX: Getting real

The goal of this chapter is to drive you through a real example of implementation and a step by step explanation.

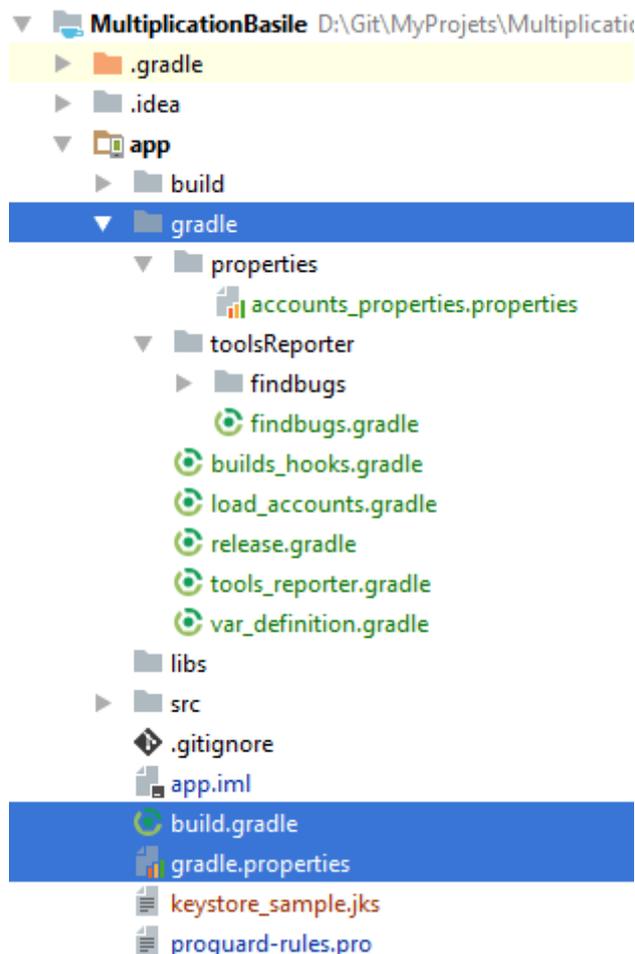
So you have read all the previous chapters, you have assimilate the notions, now let's start use them.

We presume in this chapter that you have already define your build.gradle and your project compile. We are beyond that point in this chapter; we are optimizing and enhancing it. We use a project with two dimensions of flavor (because it happens). We want to release automatically 2 applications with their reports, their tests ran, the upload to the maven central (private nexus in our case) and deployment.

18 Organize your Gradle files as your code

In you project, start by creating a "gradle_scripts" folder (or a "gradle" folder as you want, you are the one responsible for the naming, nobody cares except you and your team).

And get used to create your Gradle scripts in this folder, with sub folders hierarchy. Depending on your project, you will we quickly see the number of Gradle files increase.



So organize your files, use sub-folders and split by responsibilities. You should only let at the app sub-level the files: gradle.properties and build.gradle.

Once you have done that, let's see what to put in our gradle files.

19 Extract your build variables and constants

As usual in your build.gradle you define your "project to build the project" receipt. You should start by defining and extracting in a specific file (mine is called var_definition.gradle) all the constants and variables that you want to use in your project.

So your first line should be to define if you build a library or an application. And the second line should be to define your variables:

```
apply plugin: 'com.android.application'  
/**  
 * Our variables definition for the build script  
 */  
apply from: 'gradle/var_definition.gradle'
```

Where you are applying the gradle file that define your variable like this:

The gradle/var_definition.gradle file is the following:

```
*****  
 * Gradle definition file of our dynamic variables  
*****/  
println 'in the var_defintion'  
  
/*****  
 * Then define you attributes/variables  
 * You need to define that way for others file to know them  
 * Others gradle file can not call the method (or I didn't find yet)  
*****/  
project.ext{  
  
    versionName = "1.0.0"  
    compileSdk = 26  
    minSdk = 14  
    targetSdk = 26  
    //Can not decrease  
    versionCode = 1  
    //Gradle groupd name (for your own tasks)  
    myGradleGroup="Multiplication Basile tasks"  
  
    useSupportLibVectorDrawable = true  
  
    //build var  
    def_time_format=getDateTime()  
    def_branche_name=branch()  
    def_commit_number=commitNumber()  
    def_current_apk_name=apkCurrentSuffix()  
    def_apk_name_wrelease=apkWReleaseSuffix()  
  
    //natif tools  
    buildToolsVersion = "26.0.1"  
    supportLibVersion = "26.1.0"  
    supportAnnotationVersion = "23.1.1"  
  
    //natif  
    androidTestVersion = "1.0.0"
```

```

junitVersion = "4.12"
googlePlayServices = "11.0.1"
androidGoogleMapUtil = "0.5"
constraintLayout = "1.0.2"
supprtMultidex = "1.0.1"

//natural libs
sugarVersion = "1.4"
eventbusVersion = "3.0.0"
//debug tools
leakcanaryVersion = "1.5.1"
crashlyticsVersion = "2.6.8@aar"

//facebook analyse tools
fbStetho = "1.5.0"
//tests
mokitoVersion = "2.8.47"
espressoVersion = "3.0.0"
jsonVersion = "20160810"

//android architecture components
archiComponentVersion = "1.0.0-beta2"
}

```

Mainly we define in this file:

- The version of the SDK to use
- Variables for signing configurations
- Versions of the library we use
- Some variables used to create the file name when releasing the application.

We start our gradle.build by this definition because those variables are used in all the rest of the scripts. So if you don't define at the beginning, they are not know until you declare them. It's a good practice to define your variables at the beginning and to define them all, at the same place at the same moment.

Then we use those variables in our main gradle build script (or anywhere else):

```

android {
    compileSdkVersion project.compileSdk
    buildToolsVersion project.buildToolsVersion
    defaultConfig {
        applicationId "com.android2ee.basile.multiplication"
        minSdkVersion project.minSdk
        targetSdkVersion project.targetSdk
        versionCode project.versionCode
        versionName project.versionName
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        vectorDrawables.useSupportLibrary = project.useSupportLibVectorDrawable
    }
}

```

20 Load your external properties files

Sometimes some critical information have to be removed from the project and you extract them in a specific file (out of git watch). Then you want to load the value from this file into your gradle.

The best way is to read your properties file and doing this, to create your variables.

For the example we define our JavaKeyStore and the key to use for signing the release and the debug apk. We use the same key (not the goal to show how to sign apk):

This is the properties file used, called accounts_properties.properties:

```
#Signing configurations  
JKSFile=keystore_sample.jks  
JKSPassword=android  
JKSKeyAlias=android  
JKSKeyPassword=android
```

Then we want to load those properties and to use them to define variables that will be available in our Gradle file (or our code, as we want):

Then I create a file that will read this properties file and create the associated variables in the gradle's project object (project.myvar ou project.ext.myvar).

Extract of the file load_accounts.gradle:

```
def Properties props = new Properties()  
//find the file defining those properties  
def propFile = project.file('./gradle/properties/accounts_properties.properties')  
//You can also use rootProject  
//def propFileNotUsed = rootProject.file('./app/gradle_others.properties')  
//do a log (in case), it will be displayed every time you avluate the gradle task graph  
println "The propeties file should be ${propFile}"  
  
if (propFile.canRead()){  
    //load the properties in your Properties object  
    props.load(new FileInputStream(propFile))  
  
    //You can define specific variables of the build directly  
    if (props!=null  
        && props.containsKey('JKSFile')) {  
        //Now the properties are loaded in your props's set of variables, enjoy an reuse them  
        //here defining the debug signing config  
        //https://docs.gradle.org/current/dsl/org.gradle.api.plugins.ExtraPropertiesExtension.html  
        //To set an already existing value (but useless)  
        project.ext.set("JKSFile", file(props['JKSFile']))
```

Nothing complex here, we define a properties object, locate our properties file, load it in the properties object and then use the properties object to define our own gradle's variables.

To define gradle variables, it's just easy and you have several syntaxes. You can use the way to code you like.

```
//To create or set a value  
project.ext.set("JKSFile", file(props['JKSFile']))  
//This syntax works  
project.ext.set("JKSPassword", props['JKSPassword'])  
//This syntax works too  
project.ext.JKSKeyAlias= props['JKSKeyAlias']  
//This syntax works also  
project.ext.JKSKeyPassword=props['JKSKeyPassword']  
They all work.
```

To use those elements, it's also straight forward:

```
println "You have = project.ext.JKSPassword=${project.ext.JKSPassword}"  
println "You have = project.JKSPassword=${project.JKSPassword}"
```

And in reality, you use them in your signing block of your main gradle build file for the module:

Extract of your build.gradle file:

```
/* *****  
 * Defining and Loading Properties  
***** */  
apply from: 'gradle/load_accounts.gradle'  
/* *****  
 * Signing  
***** */  
signingConfigs {  
    debug {  
        storeFile project.ext.JKSFile  
        storePassword project.ext.JKSPassword  
        keyAlias project.ext.JKSKeyAlias  
        keyPassword project.ext.JKSKeyPassword  
    }  
    release {  
        storeFile project.ext.JKSFile  
        storePassword project.ext.JKSPassword  
        keyAlias project.ext.JKSKeyAlias  
        keyPassword project.ext.JKSKeyPassword  
    }  
}  
  
release {  
    signingConfig signingConfigs.release  
}  
debug {  
    signingConfig signingConfigs.debug  
}
```

21 Extract your hooks

From the same point of view, extract your hooks. A hook is when during the build of the gradle graph, you make an action, like ignoring specific build variants:

```
android.variantFilter { variant ->  
    if (variant.getFlavors().get(0).name.equals('basile')  
        && variant.getFlavors().get(1).name.equals('addition')) {  
        variant.setIgnore(true);  
    }  
}
```

Mainly in our hook, we:

- Ignore specific build variants
- Rename the output apk's name
- Add tests options or ignoreFailure flags

Like this:

```
/* *****  
 * Apply the build hooks here  
***** */  
/**  
 * Rename all the apk to obtain the following format:
```

```

* app-mycms_0.0.1_201704100_0101.apk
*/
android.variantFilter { variant ->
    variant.outputs.each { output ->
        output.outputFile = new File(
            output.outputFile.parent,
            output.outputFile.name.replace(".apk", def_current_apk_name + ".apk"))
    }
}

/**
 * Adapt the variants existence depending on the build you are doing 'aar or apk)
 * You don't need the same variants
 */
//Remove the BasileAddition and LilaMultiplication flavor
android.variantFilter { variant ->
    if (variant.getFlavors().get(0).name.equals('basile')
        && variant.getFlavors().get(1).name.equals('addition')) {
        variant.setIgnore(true);
    } else {
        println "Var of the project"
        println variant.getFlavors().get(0).name
        println file(STORE_FILE)
        println uploadRepo
        println "-----"
    }
    if (variant.getFlavors().get(0).name.equals('lila')
        && variant.getFlavors().get(1).name.equals('multiplication')) {
        variant.setIgnore(true);
    }
}

// Continue gradle tasks even if test fails
tasks.withType(VerificationTask) { task ->
    task.ignoreFailures = true
}

//Enable the coverage report for unit test
android.testOptions {
    unitTests.all {
        jacoco {
            includeNoLocationClasses = true
        }
    }
}

android.testOptions {
    unitTests.returnDefaultValues = true
}

```

To call this script, you just have to apply it from your main build.gradle file. But this time, as the hooks concern the end of the process. You need to wait your flavors are defined (the objects you work on), so you need to call it after this declaration:

```

/*****
 * Managing flavors
 *****/

```

```
//Give a name to your dimension
flavorDimensions "enfants", "operator"

//define your flavors (as one flavor has a dimension they must all have one)
productFlavors { //blabla flavors description
}

/*****
* Apply the build hooks here
*****/
apply from: 'gradle/builds_hooks.gradle'
```

22 Building, testing and Analyzing Script

To create a release script you will need to aggregate several tasks:

- Build the project
- Run the tests (Unit and Android)
- Run the analyzer and gather the reports
- Move the deliverables to the delivery folder
- Upload the application (or archive) to your Nexus
- Deploy the application on GooglePlay (but not explain in this article)

Let's create those elements steps by steps. We want first a task that build and check the result of the build.

22.1 BuildAndCheckProject task

This task is simple, it's an entry point and you just want it to run the task build and check. It looks like something like this:

```
/**
 * Run the build, the tests and the analyzers tools reports
 */
task buildAndCheckProject(dependsOn: [
    'fullBuild', //ok
    'runReportersTools'
]) {
    group = project.ext.myGradleGroup
    description = "Run full build and then the full check."
    doFirst {
        println 'Starting the build'
    }
    doLast {
        println 'Full complete check of MyCms application is done'
    }
}
```

Yep, we just say what we depend on two tasks, fullBuild and runReportersTools. You need to synchronize those two tasks, because you need to run your checking AFTER the build is done and the tests are passed so at the end of the file you just explain that to your script:

```
runReportersTools.mustRunAfter ':app:fullBuild'
```

Those tasks are not defined yet, this is the goal of the next chapters.

22.2 fullBuild task

The task fullBuild is dedicated to create the applications and run the tests on it. In the example project there is no smart tests yet, perhaps one day they will come.

So full build, in a way does nothing, it just depends on 3 tasks:

- ':app:build',
- ':app:connectedAndroidTest',
- ':app:test'

Those tasks are native tasks of your project.

So all you have to do is:

```
/**
 * Build then Run AndroidTests and UnitTests.
 */
task fullBuild(dependsOn: [':app:build', ':app:connectedAndroidTest', ':app:test']) {
    group = project.ext.myGradleGroup
    description = "Build then Run AndroidTests and UnitTests."
    doLast {
        println 'Done'
    }
}
```

22.3 runReporters task

This task is launching several tasks; one task for a specific tool. The several tools we want to use:

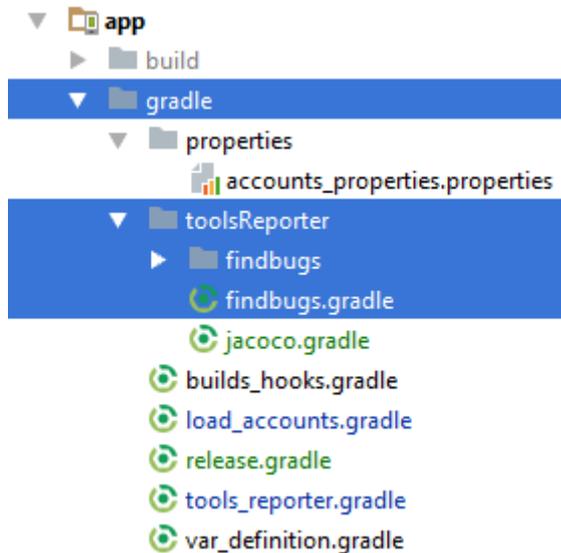
- FindBugs
- Jacoco
- PMD
- Lint
- Unit tests results
- Android tests results
- Javadocs

In fact tests results are already created by the FullBuild, you just want to move them.

```
/**
 * Generate the Analysers tools reports on your projects
 */
task runReportersTools(dependsOn: [
    'findbugs',//ok
    'jacocoTestReport',//ok
    'pmd',//ok
    'androidJavadocs',//ok
    //infer is missing (troubles on windows)
    //lint is done by the native android build script
]) {
    group = project.ext.myGradleGroup
    description = "Generate the Tools analyzers reports."
    doFirst {
        println 'LaunchingCheck'
    }
    doLast {
        //copy your report folder to the release folder
        moveTheReportToReleaseFolder
        println 'check done'
    }
}
```

```
}  
}
```

In fact we are going to inject libraries in our gradle script. All those libraries scripts will be related with reporters. It's the moment to create a folder:



In the toolsReporter folder we gather all those scripts.

22.3.1 moveTheReportToReleaseFolder task

It's just a task that copy paste content from one folder to another one. This type of task is current in gradle scripts.

```
/*  
*****  
* Move your report to your release folders  
*****  
*/  
  
//Move the generated files to the delivery  
task moveTheReportToReleaseFolder(type: Copy,  
    dependsOn: ['fullBuild', 'cleanReleaseReportDirectory']) {  
    group = project.ext.myGradleGroup  
    description = "Move the build/report folder into the weekly_release."  
    println "Task moveTheReportsToReleaseFolder"  
    from 'build/reports/'  
    into 'weekly_release/' + def_branche_name + '/reports/'  
    include('**/*')  
    exclude('**/*-release-*')  
    doLast {  
        println 'moveTheFileToReleaseFolder over'  
    }  
}  
  
//Remove directory where release test reports will be copied into  
task cleanReleaseReportDirectory(type: Delete) {  
    group = project.ext.myGradleGroup  
    description = "Clean the directory of the report in the weekly_release."  
    doLast {  
        delete 'weekly_release/' + def_branche_name + '/reports/'  
    }  
}
```

22.4 runReporters:FindBug task

We start by include FindBugs.

First, we need to retrieve the gradle plugin itself. You need to add those repository to your artefact repository list, in your main build.gradle (the one of the project, not the one of your module):

```
buildscript {
    repositories {
        mavenCentral()
        jcenter()
    }
    dependencies {
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
        mavenCentral()
    }
}
```

Second, you need to create your gradle script file to run the library, let's call it findbugs.gradle under toolsReporter folder:

```
apply plugin: 'findbugs'
/**
 * Define your directory
 */
def reportsDir = "${project.buildDir}/reports"

task findbugs(type: FindBugs, dependsOn: [
    'app:assembleBasileMultiplicationDebug',
    'app:assembleLilaAdditionDebug',
    'app:cleanPreviousFindbugsReports',//ok
]) {
    group = project.ext.myGradleGroup
    description = "Generate the findbugs report."
    ignoreFailures = true
    reportLevel = "low"
    effort = "max"
    excludeFilter = new File("gradle/toolsReporter/findbugs/findbugs-filter.xml")
    //Define exactly where are the class in your Build folder to analyze
    classes = files("${buildDir}/intermediates/classes/basileMultiplication",
        "${buildDir}/intermediates/classes/lilaAddition")

    //Where are your sources:
    source 'src/main/java', 'src/basile/java', 'src/lila/java', 'src/addition/java', 'src/multiplication/java'
    include '**/*.java'
    exclude '**/gen/**'

    reports {
        xml.enabled = false
        html.enabled = true
        xml {
```

```

        destination new File("$reportsDir/findbugs/findbugs.xml")
    }
    html {
        destination new File("$reportsDir/findbugs/findbugs.html")
    }
}

classpath = files()
}

```

First notice we start by applying the plugin findbugs.

Then we define a task called findBugs:

```

task findbugs(type: FindBugs, dependsOn: [
    ':app:assembleBasileMultiplicationDebug',
    ':app:assembleLilaAdditionDebug',
    ':app:cleanPreviousFindbugsReports', //ok
]) {

```

This task is a FindBugs type task (the one defined in the plugin findbugs we imported) and depends on assemble.

The FindBugs task needs several parameters to work fine:

- Which file to exclude from the analysis
- Where is the classes (your bytecode to analyze, in your build folder)
- Where is your source class (your code in fact)
- What is the pattern of the file to include, the one to exclude
- What type of report you want and where
- Which is the min level you want to use to detect bugs
- Which is the effort to make
- If you fail when findbugs detects errors

And the file above is just the syntax to use to provide those parameters to the task.

22.4.1 The findbugs-filter.xml file

This is a configuration file for findBugs, (documentation can be found here:

<http://findbugs.sourceforge.net/manual/filter.html>). It is localized under toolsReporter/findbugs.

```

<?xml version="1.0" encoding="UTF-8"?>
<FindBugsFilter>
  <!-- http://stackoverflow.com/questions/7568579/eclipsefindbugs-exclude-filter-files-doesnt-work -->
  <Match>
    <Class name="~.*\R$.*" />
  </Match>
  <Match>
    <Class name="~.*\Manifest$.*" />
  </Match>
  <!-- All bugs in test classes, except for JUnit-specific bugs -->
  <Match>
    <Class name="~.*\.*Test" />
    <Not>
      <Bug code="IJU" />
    </Not>
  </Match>

```

</FindBugsFilter>

22.5 runReporters:Jacoco task

Jacoco is a test coverage tool.

First, we need to retrieve the gradle plugin itself. You need to add those repository to your artefact repository list, in your main build.gradle (the one of the project, not the one of your module):

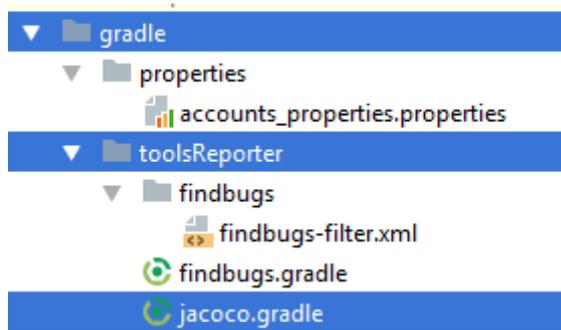
```
buildscript {
    repositories {
        mavenCentral()
        jcenter()
    }
    dependencies {
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
        classpath 'com.android.tools.build:gradle:3.0.0-beta2'
        //Gradle check tasks : Jacoco
        classpath 'org.jacoco:org.jacoco.core:0.7.7.201606060606'
    }
}
```

Second step, disable your default test coverage for your build gradle

Extract from build.gradle

```
buildTypes {
    release {
        signingConfig signingConfigs.release
        minifyEnabled false
        shrinkResources false
        useProguard false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        //add tests coverage using Jacoco
        testCoverageEnabled false
    }
    debug {
        signingConfig signingConfigs.debug
        applicationIdSuffix '.debug'
        versionNameSuffix '.debug'
        buildConfigField("boolean", "isallowed", "true")
        resValue "string", "hidden_string", "I love debug"
        //add tests coverage using Jacoco
        testCoverageEnabled false
    }
}
```

Third, you need to create your gradle script file to run the library, let's call it jacoco.gradle under toolsReporter folder:



The task itself follows the same principles as the ones for findBugs, you apply the plugin, your task is of a specific type and you give the parameters to the “parent” task.

It looks like something like that:

apply **plugin: 'jacoco'**

```

/*****
 * Create Jacoco Report for each application you release
 *****/
/**
 * Generate the JaCoco report for Basile Apk
 */
task jacocoTestReportBasile(type: JacocoReport, dependsOn: [
    //you depend on nothing because it's called by gWR...but
    //if you want to just generate your reports from scratch uncomment those lines
    // ':app:assembleBasileMultiplicationDebug',
]) {

    reports {
        xml.enabled = true
        html.enabled = true
        xml {
            destination new File("$reportsDir/jacocoBasile/jacocoReportBasile.xml")
        }
        html {
            destination new File("$reportsDir/jacocoBasile/jacocoReportBasile.html")
        }
    }

    //Define exactly where are the class in your Build folder to analyze
    def debugTreeBasile = fileTree(dir: "${buildDir}/intermediates/classes/basileMultiplication/release",
        excludes: fileFilter)
    def mainSrc = files(["src/main/java",
        "src/basile/java",
        "src/addition/java",
        "src/multiplication/java"])

    //Where are your sources:
    sourceDirectories = files([mainSrc])
    classDirectories = files([debugTreeBasile])
    //As you want to gather all your tests reports, add the ec and exec you want to be took into
    //account when generating the report
    executionData = fileTree(dir: "$buildDir", includes: [
        "jacoco/testBasileMultiplicationDebugUnitTest.exec",
        "jacoco/testBasileMultiplicationReleaseUnitTest.exec",
        "outputs/code-coverage/connected/flavors/**/*coverage.ec"
    ])
}

```

```
    )
}
```

So mainly, for a JacocoReport task you need to provide the following parameters:

- Which and where you want your reports
- Where is the bytecode you want to analyze with the file name patterns you want to exclude from your analysis
- Where is your source code
- And only tricky part, where are your execution data

So the only point is the execution data, right ?

```
executionData = fileTree(dir: "$buildDir", includes: [
    "jacoco/testBasileMultiplicationDebugUnitTest.exec",
    "jacoco/testBasileMultiplicationReleaseUnitTest.exec",
    "outputs/code-coverage/connected/flavors/**/*coverage.ec"
])
```

So, you have to know that when we are running our test and analyze them, jacoco generates files that are which and how tests have been ran on your code. When you parametrize a Jacoco task, you need to provide the file you want to analyze. Mainly, for Android Tests, they are generated under jacoco folder (in your build folder) with the exec extension. For Unit Tests, you have them under outputs/code-coverage/connected/flavors with the extension ec. Find the file you are interested with and use them for your analysis.

This is the way to gather several type of tests into the same code coverage report using Jacoco. Here we use the tests from AndroidTest and unit tests to create the final code coverage report including all those tests. This is an important point.

22.5.1 Jacoco with several flavors and product

If you deliver several application or archives with one project, because you play with flavors, you will want to have a specific report for each apk (or aar). To do that you need to create a specific task by apk and gather them like this:

```

/*****
 * Main task
 *****/
task jacocoTestReport( dependsOn: [
    ':app:jacocoTestReportBasile',
    ':app:jacocoTestReportLila',
])
/*****
 * Create Jacoco Report for each application you release
 *****/
/**
 * Generate the JaCoco report for Basile Apk
 */
task jacocoTestReportBasile(type: JacocoReport, dependsOn: [
    //you depend on nothing because it's called by gWR...but
    //if you want to just generate your reports from scratch uncomment those lines
    // ':app:assembleBasileMultiplicationDebug',
]) {

    reports {
        xml.enabled = true
    }
}

```

```

    html.enabled = true
    xml {
        destination new File("$reportsDir/jacocoBasile/jacocoReportBasile.xml")
    }
    html {
        destination new File("$reportsDir/jacocoBasile/jacocoReportBasile.html")
    }
}

//Define exactly where are the class in your Build folder to analyze
def debugTreeBasile = fileTree(dir: "${buildDir}/intermediates/classes/basileMultiplication/release",
    excludes: fileFilter)
def mainSrc = files(["src/main/java",
    "src/basile/java",
    "src/addition/java",
    "src/multiplication/java"])

//Where are your sources:
sourceDirectories = files([mainSrc])
classDirectories = files([debugTreeBasile])
//As you want to gather all your tests reports, add the ec and exec you want to be took into
//account when generating the report
executionData = fileTree(dir: "$buildDir", includes: [
    "jacoco/testBasileMultiplicationDebugUnitTest.exec",
    "jacoco/testBasileMultiplicationReleaseUnitTest.exec",
    "outputs/code-coverage/connected/flavors/**/*coverage.ec"
])
}

/**
 * Generate the JaCoco report for Lila Apk
 */
task jacocoTestReportLila(type: JacocoReport, dependsOn: [
    //you depend on nothing because it's called by gWR...but
    //if you want to just generate your reports from scratch uncomment those lines
    // ':app:assembleLilaAdditionDebug',
]) {
    reports {
        xml.enabled = true
        html.enabled = true
        xml {
            destination new File("$reportsDir/jacocoLila/jacocoReportLila.xml")
        }
        html {
            destination new File("$reportsDir/jacocoLila/jacocoReportLila.html")
        }
    }
}

//Define exactly where are the class in your Build folder to analyze
def debugTreeLila = fileTree(dir: "${buildDir}/intermediates/classes/lilaAddition/release",
    excludes: fileFilter)
def mainSrc = files(["src/main/java",
    "src/lila/java",
    "src/addition/java",
    "src/multiplication/java"])

//Where are your sources:

```

```

sourceDirectories = files([mainSrc])
classDirectories = files([debugTreeLila])
//As you want to gather all your tests reports, add the ec and exec you want to be took into
//account when generating the report
executionData = fileTree(dir: "$buildDir", includes: [
    "jacoco/testLilaAdditionDebugUnitTest.exec",
    "jacoco/testLilaAdditionEleaseUnitTest.exec",
    "outputs/code-coverage/connected/flavors/**/*coverage.ec"
])
}

```

You have your entry task that launch, for each application, the associated Jacoco task, with its specific parameters.

22.6 runReporters: Javadoc task

As usual for those tasks, in your tools_reporter.gradle file, you apply your gradle file containing your Javadoc task, like this

apply from: 'gradle/toolsReporter/javadoc.gradle'

```

/*****
 * Your Tasks for building and checking
 *****/
/**
 * Generate the Analysers tools reports on your projects
 */
task runReportersTools(dependsOn: [
    'findbugs',//ok
    'jacocoTestReport' //ok
    'generateProjectJavadocs' //ok even if the tasks fails, javadoc are generated
    //but if not declared no javadoc generated

```

I spend some times trying to make this task running without problem, but I failed. All I can do is generating the javadocs but the task failed with errors. Most of the time it doesn't have the reference to outside classes. By the way, so it failed, but it does enough.

So then the file Javadoc.gradle, is like usual, in a way:

```

/*****
 * Creating the Javadocs
 * https://docs.gradle.org/current/dsl/org.gradle.api.tasks.javadoc.Javadoc.html
 *****/
task generateProjectJavadocs(type: Javadoc) {
    group = project.ext.myGradleGroup
    description = "Generate the JavaDoc of the project."
    //because it will fail but generate your JavaDoc (probability of failure 80%, javadoc generation 100% sure)
    failOnError false
    title = "Javadoc of the Project Multiplication"
    source 'src/main/java', 'src/basile/java', 'src/lila/java', 'src/addition/java', 'src/multiplication/java'
    classpath = files(project.android.getBootClasspath())
    destinationDir = file("${buildDir}/reports/javadoc/")

    exclude '**/BuildConfig.java'
    exclude '**/R.java'
    doFirst(){
        println("Generating the Javadocs is starting")
    }
}

```

```

doLast(){
    println("Generating the Javadocs is done")
}
}

```

As for the others tasks of that type, you need to provide information. Here the task needs to know:

- Where is the source
- Where is the destination
- Which file to exclude

We also add the flag failOnError to false to keep going with our gradle build even if Javadoc find a problem (like where is android.View class).

22.7 runReporters: PMD task

Really the same pattern we apply:

In your tools_reporter.gradle file you add PMD and call it in the runReportersTools task:

```

apply from: 'gradle/toolsReporter/pmd.gradle'
//apply from: "gradle/check/codequality-infer.gradle"
//apply from: "gradle/check/codequality-checkstyle.gradle"

/*****
 * Your Tasks for building and checking
 *****/

/**
 * Build then Run AndroidTests and UnitTests.
 */
task fullBuild(dependsOn: [':app:build', ':app:connectedAndroidTest', ':app:test']) {
    group = project.ext.myGradleGroup
    description = "Build then Run AndroidTests and UnitTests."
    doLast {
        println 'Done'
    }
}

/**
 * Generate the Analysers tools reports on your projects
 */
task runReportersTools(dependsOn: [
    'findbugs', //ok
    'jacocoTestReport', //ok
    'pmd', //ok
    'generateProjectJavadocs' //ok even if the tasks fails, javadoc are generated

```

Now you create your file script,

And you create your PMD task in it:

```

apply plugin: 'pmd'

/*****
 * PMD file
 *****/
def reportsDir = "${project.buildDir}/reports"

// Add checkstyle, findbugs, pmd and lint to the check task.

```

```

check.dependsOn 'pmd'

/**
 * Launch PMD on the project
 */
task pmd(type: Pmd, dependsOn: [
    ':app:assembleDebug',
    ':app:cleanPreviousPmdReports' //ok
]) {
    group = "MyCMS_tasks"
    description = "Generate the pmd report, but I am not sure we want it."
    ignoreFailures = true
    ruleSetFiles = files("gradle/toolsReporter/pmd/pmd-ruleset.xml")
    ruleSets = []

    source 'src/main/java', 'src/basile/java', 'src/lila/java', 'src/addition/java', 'src/multiplication/java'
    include '**/*.java'
    exclude '**/gen/**'

    reports {
        xml.enabled = false
        html.enabled = true
        xml {
            destination "$reportsDir/pmd/pmd.xml"
        }
        html {
            destination "$reportsDir/pmd/pmd.html"
        }
    }
}

//Remove directory where release test reports will be copied into
task cleanPreviousPmdReports(type: Delete) {
    doLast {
        delete "${project.buildDir}/reports/pmd"
    }
}

```

Really as usual. You apply the plugin you want to use, here pmd, and you give it the parameters, source, report types, reports destination.

The only main difference is the rule set. I create the file under ./pmd/ and its content is the default one:

```

<?xml version="1.0"?>
<ruleset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="Android Application Rules"
  xmlns="http://pmd.sf.net/ruleset/1.0.0"
  xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd"
  xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0 http://pmd.sf.net/ruleset_xml_schema.xsd">
  <description>Custom ruleset for Android application</description>

  <exclude-pattern>.*R.java</exclude-pattern>
  <exclude-pattern>.*gen/*</exclude-pattern>

  <rule ref="rulesets/java/android.xml" />
  <rule ref="rulesets/java/clone.xml" />
  <rule ref="rulesets/java/finalizers.xml" />
  <rule ref="rulesets/java/imports.xml">
    <!-- Espresso is designed this way !-->

```

```

    <exclude name="TooManyStaticImports" />
</rule>
<rule ref="rulesets/java/logging-java.xml" />
<rule ref="rulesets/java/braces.xml" />
<rule ref="rulesets/java/strings.xml" />
<rule ref="rulesets/java/basic.xml" />
<rule ref="rulesets/java/naming.xml">
    <exclude name="AbstractNaming" />
    <exclude name="LongVariable" />
    <exclude name="ShortMethodName" />
    <exclude name="ShortVariable" />
    <exclude name="VariableNamingConventions" />
</rule>
</ruleset>

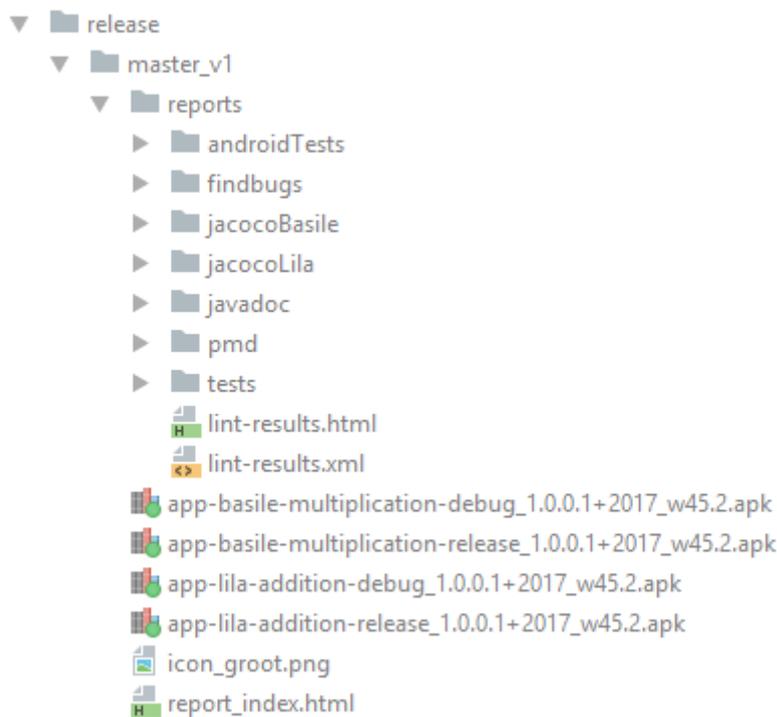
```

<http://pmd.sourceforge.net/pmd-5.1.1/howtomakearuleset.html>

<https://pmd.github.io/pmd-5.8.1/pmd-java/rules/index.html>

23 The release scripts

We are ready to create the release. The goal, press a button and have every think generated for you, like this:



In your release folder, you want:

- All your apk
- your lint, androidTests,findbugs,jacoco,pmd,tests reports
- your Javadoc

So what we want to do is simple:

apply from: 'gradle/release/to_release_folder.gradle'

```
//This is the root task
task generateRelease(dependsOn: ['clean',
    'cleanReleaseReportDirectory',
    'packageRelease',
    'releaseBuild']) {
    group = project.ext.myGradleGroup
    description = "Generate the Release."
    println "Task generateRelease processing..."
    //I do nothing, In am an entry point
    //I use task dependency to launch the build
    doLast{
        println 'upload is next step...' }
    }
}
```

The only not yet explained task is packageRelease. In fact it moves around the files from somewhere to your release folder. We have defined those tasks in the release/to_release_folder.gradle file.

This file has 5 gradle tasks:

- packageRelease
- moveTheApkToReleaseFolder
- moveTheReportToReleaseFolder
- moveTheReportIndexToReleaseFolder
- cleanReleaseReportDirectory

This file is:

```
/******
 * Task to move report/apk/aar to the release folder
 *****/
def releaseRootFolder='release/' + def_branche_name+'_v'+project.ext.versionCode
/**printing content of the release directory
//http://stackoverflow.com/questions/18569983/androidgradle-list-directories-into-a-file
//The task name is normally packageRelease
//But for this file, the following name is better*/
task packageRelease(dependsOn: ['moveTheReportToReleaseFolder',
    'moveTheReportIndexToReleaseFolder'',
    'moveTheApkToReleaseFolder'']) {
    group = project.ext.myGradleGroup
    description = "List the file that belongs to the release folder. Depends on the fullBuild and move tasks."
    doFirst {
        println 'Task packageRelease'
    }
    doLast {
        //then list them
        println 'starting the work with the branch' + def_apk_name_wrelease
        println "Listing files contained in: ${releaseRootFolder} == "
        //define the variables for files location
        def releaseRootF =new File(releaseRootFolder)
        releaseRootF.mkdirs()
        releaseRootF.setWritable(true)
        def listFile = new File(releaseRootF, 'FolderContent.txt')
        listFile.setWritable(true)
        //define the content to write
        def contents = ""
        //get the files tree
```

```

def tree = fileTree(dir: releaseRootF, excludes: ['**/*.js',
                                                '**/*.css',
                                                '**/*.html',
                                                '**/*.xml'])

//browse all those files
tree.visit { fileDetails ->
    if (!fileDetails.isDirectory()) {
        contents += "${fileDetails.relativePath}" + "\n"
    }
}
//write the content
listFile.write contents
println 'listReleaseFiles over'
}

/**
 * Move the generated files to the delivery
 */
task moveTheApkToReleaseFolder(type: Copy,
    dependsOn: 'fullBuild') {
    group = project.ext.myGradleGroup
    description = "Move generated apk into the release_folder inside the good folder (depend on the branch).
It depends on the "
    //you need to define the xact folder, else it reproduces the folder hierarchy
    from 'build/outputs/apk/basileMultiplication/debug',
        'build/outputs/apk/lilaAddition/debug',
        'build/outputs/apk/basileMultiplication/release',
        'build/outputs/apk/lilaAddition/release'
    into releaseRootFolder
    include('**/*.apk')
    //I want all the apk, but if you want only debug, you have them below:
    // exclude('**/*-release-*.apk')
    rename { String fileName ->
        fileName.replace(".apk", def_apk_name_wrelease + ".apk")
    }
    doFirst {
        println 'Task moveTheFileToReleaseFolder'
    }
    doLast {
        println 'moveTheFileToReleaseFolder over'
    }
}

//Move the generated files to the delivery
task moveTheReportToReleaseFolder(type: Copy,
    dependsOn: ['runReportersTools',
                'cleanReleaseReportDirectory']) {
    group = project.ext.myGradleGroup
    description = "Move the build/report folder into the weekly_release."
    from 'build/reports/'
    into releaseRootFolder + '/reports/'
    include('**/*')
    exclude('**/*-release-*.apk')
    doFirst {
        println 'Task moveTheReportsToReleaseFolder'
    }
    doLast {
        println 'moveTheFileToReleaseFolder over'
    }
}

```

```

}

//Move the report index to the delivery
task moveTheReportIndexToReleaseFolder(type: Copy) {
    group = project.ext.myGradleGroup
    description = "Move the build/report index into the weekly_release."
    from 'gradle/release/report'
    into releaseRootFolder
    include('**/*')
    exclude('**/*-release-*')
    filter { String line ->
        if(line.contains('${DateOfTheDay}')){
            line.replace('${DateOfTheDay}',project.ext.def_readableSchoolDate)
        }else line
    }
    doFirst {
        println 'Task moveTheReportsIndexToReleaseFolder'
    }
    doLast {
        println 'moveTheFileToReleaseFolder over'
    }
}

//Remove directory where release test reports will be copied into
task cleanReleaseReportDirectory(type: Delete) {
    group = project.ext.myGradleGroup
    description = "Clean the directory of the report in the weekly_release."
    delete releaseRootFolder + '/reports',
        releaseRootFolder + '/report_index.html'
}
runReportersTools.mustRunAfter 'cleanReleaseReportDirectory'
moveTheReportIndexToReleaseFolder.mustRunAfter 'cleanReleaseReportDirectory'

```

23.1 One entry page for your report

A good practice is to have one html entry point to browse your tests and reports. I reuse the one of lint to obtain this one:

Tools analysis reports for the project MultiplicationBasile

Tests Reports

- [Unit Tests Basile Debug](#)
Is the tests associated with the Basile apk (debug config).
- [Unit Tests Basile Release](#)
Is the tests associated with the Basile apk (release config).
- [Unit Tests Lila Debug](#)
Is the tests associated with the Lila apk (debug config).
- [Unit Tests Lila Release](#)
Is the tests associated with the Lila apk (release config).
- [Basile Instrumented Tests](#)
Instrumented tests for Basile apk.
- [Lila Instrumented Tests](#)
Instrumented tests for Lila apk.

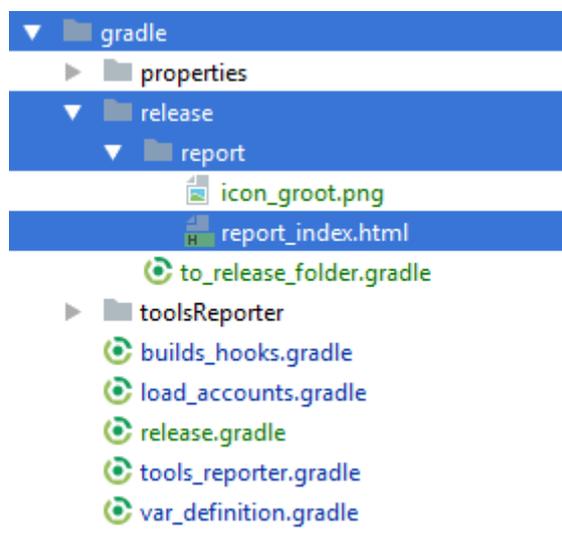
For MyCms Android testing strategy, follow the links

Tools analysis Reports

- [Lint Full Report](#)
Official Android coding rules tool for Android provided by Google
- [FindBugs](#)
FindBugs is a program which uses static analysis to look for bugs in Java code.
- [Basile Tests coverage](#)
Tests coverage, all is said in the title.

It's pretty easy and will save you a lot of time if you want to use your reports to analyze your project.

I just created a folder with the root page:



And we create a task to copy paste this file into the release folder. We add the date of the generation of the report and the version of the application in the html file.



The task is easy:

```
task moveTheReportIndexToReleaseFolder(type: Copy) {
    group = project.ext.myGradleGroup
    description = "Move the build/report index into the weekly_release."
    from 'gradle/release/report'
    into releaseRootFolder
    include('**/*')
    exclude('**/*-release-*')
    filter { String line ->
        if(line.contains('${DateOfTheDay}')){
            line.replace('${DateOfTheDay}',project.ext.def_readableSchoolDate)
        }else line
    }
    doFirst {
        println 'Task moveTheReportsIndexToReleaseFolder'
    }
    doLast {
        println 'moveTheFileToReleaseFolder over'
    }
}
```

The tricky part is the filter, which is in fact call each time a line is found, you can manipulate the string. And it's related to this html extract (in the file report_index):

```
<header class="mdl-layout__header">
  <div class="mdl-layout__header-row">
    <span class="mdl-layout-title">
      Tools analysis reports for the project MultiplicationBasile</span>
    <div class="mdl-layout-spacer"></div>
    <nav class="mdl-navigation mdl-layout--large-screen-only">
      Automatically generated the ${DateOfTheDay}
      <br>
      BasileMultiplication exact version name: ${VersionNumber}
    </nav>
  </div>
</header>
```

You just have to provide a good html file to present your report. In this html file you just have to use relative path (you are in your release folder).Like this:

```
<h2 class="mdl-card__title-text">
  <a href="/reports/tests/testLilaAdditionDebugUnifTest/index.html">
    Unit Tests Lila Debug</a>
</h2>
Is the tests associated with the Lila apk (debug config).
<br><br>
```

24 The upload part

What's left to do according to the release? Two elements; upload on a repository (maven repository) and deliver the application.

I use private Nexus Repository and Balto for that. So useless for us in a generic way.

For automatically deliver to GooglePlay, I have found this plugin that seems to be really easy to use:

<https://github.com/Triple-T/gradle-play-publisher>

For the upload to mavenCentral, JForg or Nexus, you always will have the same information to provide.

24.1 Upload: Maven basics principles

When uploading your project to a MavenCentral repo, you go back to Maven concepts. Gradle has been built upon Maven (which has been created upon Ant...) and it really helps us. A while ago (7 years), I wrote a book explaining how to use Maven and Jenkins for CI on Android. Well almost a prehistoric book now in 2017, but at the time, it was hell just to make Eclipse, Maven and the Android project build worked together. Today, I can tell you, each morning you can send a "I love you" to the Gradle team and the Gradle community, those silent actors that have enhanced our daily life so concretely. Hey dudes, thank.

So I want to share with you the basics on Maven, I copy paste what I was explaining 7 years ago. It's really important to have those concepts in mind when trying to talk with a Maven repository.

[Extract from "Android A Complete Course" M. Seguy android2ee@2010]

Let's stop thinking « Android » for a while and start thinking « Java » again.

Maven's goal is to rationalize a project's life cycle by means of tools and norms. This means managing compilation, setting unitary tests, managing packaging, deployment, project reports (java doc, test reports, PMD), linking to a source versioning tool, linking to a continuous integration tool...

That is why Maven provides tools and norms.

The tools will write certain files for you (MakeFile, Ant). The normalization allows any developer to go from one project to another without losing sight of the project. The POM file (Project Object Model) is a normalized file that describes the project and enables Maven to follow its life cycle.

Let's start by the beginning and build a Maven project:

1.1.1.1 First Pom

Create a project called *MyProject*

Create a pom.xml file in MyProject and write in this content:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.myproject</groupId>
  <artifactId>MyProject</artifactId>
  <version>SNAPSHOT</version>
</project>
```

Create the folder src/main/java/com/mycompany/myproject in MyProject

Write the source src/main/java/com/mycompany/myproject/Person.java in

```
package com.mycompany.myproject;
public class Person {
    public String name;
    public String telephone;
    public String email;
    public String toString() {
        return new StringBuilder()
            .append(name).append(",")
            .append(telephone).append(",")
            .append(email).toString();
    }
}
```

Compiler et packager: Do "mvn package" in the MyProject folder

```
[INFO] Building jar: /MyProject/target/MyProject-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
```

The projects' sources are compiled in target/classes.

The project is packaged in target/MyProject-SNAPSHOT.jar

1.1.1.2 What just happened?

1.1.1.2.1 Project coordinates

Maven provides a coordinates system for all projects. These coordinates have three variables (just like with latitude, longitude, altitude) that are:

groupId: This is the group identifier (for an organization, team, company or any other kind of development group). Conventionally, start with the domain name of the organization (in the same way as with the project's root java package) which can be com, net, org... Example net.fr.tlse.sti.android

artifactId: Identifier of the project in the group of projects that belong to groupId. Do not use any « . » in the names of the artifactId

version: Identifier of the version to build. There are two different types: the first type is a normal version number, the other type is a Snapshot. By convention for a normal version number, the identifier should be:

<major version>.<minor version>.<incremental version>-<qualifier>

This, for example, would give us 2.1.0-alpha. The qualifier is not mandatory, the other elements are highly recommended. The Snapshot version type is in fact a kind of syntax that allows you to build versions (typically during the over-night build) that are named in function of the date on which the build took place. This syntax is to be used in a project under development that has not yet reached a version but is performing builds on a regular basis (either with continuous integration, or by opening up the source for other modules or projects).

There is also a fourth identifier:

Classifier: Allows you to add to coordinate (so to a same project with a same version i.e. the same code) a piece of additional information. This might be the type of JDK used by the build, the destination platform... It is a way to add information to a code that is the same but not packaged in the same way. In other words, a project that is compiled with the JDK 1.3 and the JDK 1.6 without any modifications in the code will use classifier to distinguish between the two builds.

So my first POM defines the project (its unique and exact name).

1.1.1.2.2 Project structure convention

By convention, Maven assumes that a project is structured in the following way:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   |-- com
    |   |   |   |-- mycompany
    |   |   |   |   |-- app
    |   |   |   |   |   |-- App.java
    |   |-- resources
    |   |   |-- com
    |   |   |   |-- mycompany
    |   |   |   |   |-- app
    |   |   |   |   |   |-- App.properties
    |-- test
    |   |-- java
    |   |   |-- com
    |   |   |   |-- mycompany
    |   |   |   |   |-- app
    |   |   |   |   |   |-- AppTest.java
    |-- resources
    |   |-- com
    |   |   |-- mycompany
    |   |   |   |-- app
    |   |   |   |   |-- AppTest.properties
```

This way, Maven does not need any additional information to compile, package or test a project. By respecting this structure you allow Maven to perform actions on the project. It is for this reason that the class has been put in the directory `src/main/java/com/mycompany/myproject`.

When working with Android, it is impossible to respect this structure because it clashes with the imposed structure of an Android project. It's up to Maven to adapt, not Android.

[Note from 2017: At the time, it was really a hell, not possible to be compatible with the maven default structure. It was tough, I swear, it was really tough]

1.1.1.2.3 A build project's life cycle

For Maven, projects are always built step by step. Each phase being necessary to perform the one following it. Each stage can be composed of several sub-tasks, goal. The list of tasks for a build is:

Validate: Validates that the project is correct and that all the necessary information is present.

Compile: Compiles the source code of the project.

Test: Tests the compiled source code by using the available test framework. These tests do not require that the code be deployed or packaged.

Package: Packages the compiled code in its targeted distribution format (Jar, War, Apk).

Integration-test : Generates and deploys the package in an integration space when necessary and performs integration tests.

Verify: Launches the verifications that makes sure that the package is valid and that it meets quality criteria.

Install: Installs the package in the local repository so it can be used as a dependency for local projects that reference it.

Deploy: This is done in an integration or production space, it copies the final package in the remote repository which allows it to be shared with other developers and other projects.

For Maven to perform a task, simply tell it to do so. Hence, to launch:

- compilation: « mvn: package » is sufficient,
- tests, « mvn:verify » is sufficient,
- sharing with other projects, « mvn:install » must be called (the package is deployed in the local repository).

Each phase has a certain number of goals. Each phase is linked to a principal goal, which is called during mvn:***. This mapping has been partially reproduced here:

Phases	Goals
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

[End of Extract from "Android A Complete Course" M. Seguy android2ee@2010]

24.2 Upload : In a Nexus Repository

We use the Maven plugin to do that. So at the top of your gradle.build, apply it:

```
/**
```

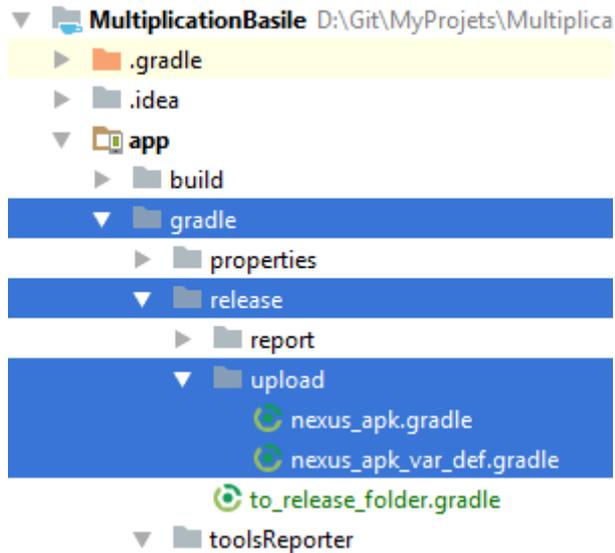
```
* Used to deploy on Nexus our apk
```

*/
apply plugin: 'maven'

More resources on the plugin: https://docs.gradle.org/current/userguide/maven_plugin.html

To upload in a Nexus, you will need two elements. The script in one file, the parameters in an other one. The first one is shared with the team, the second one is confidential. Like you did for your Signing configurations. This way you insure the internal ship cannot make the biggest mistake of his stage.

So create your files in your gradle/upload/nexus or gradle/upload folder.



Let's start by the properties you need. The file nexus_apk_var_def.gradle define them like this:

You will need all the value below:

```
//Define the coordinates of your project
//-----
project.ext {
//Describe the coordinates of your project (optional)
    POM_NAME = "BasileMultiplication Application deployed on Nexus"
//groupId: This is the group identifier (for an organization, team, company or any other kind of
//development group). Conventionally, start with the domain name of the organization (in the same
//way as with the project's root java package) which can be com, net, org... Example
    POM_GROUP_ID = "com.android2ee.basile.multiplication"
//artifactId: Identifier of the project in the group of projects that belong to groupId. Do not use any
//« . » in the names of the artifactId
//Exemple: acms-android-mycms or acms-android-androidauto
    POM_ARTIFACT_ID = "basile-multiplication"
//version: Identifier of the version to build. There are two different types: the first type is a normal
//version number, the other type is a Snapshot. By convention for a normal version number, the
//identifier should be:
//<major version>.<minor version>.<incremental version>-<qualifier>
//This, for example, would give us 2.1.0-alpha. The qualifier is not mandatory, the other elements are
//highly recommended. The Snapshot version type is in fact a kind of syntax that allows you to build
//versions (typically during the over-night build) that are named in function of the date on which the
//build took place. This syntax is to be used in a project under development that has not yet reached a
//version but is performing builds on a regular basis (either with continuous integration, or by opening
```

```

//up the source for other modules or projects).
VERSION_NAME = project.def_apk_name_wrelease
//Tell if it is a release or not
//Release version needs to be sign using a gpg (bottom of the file)
//It can be TRUE or FALSE
RELEASE_VERSION = "FALSE"
//The packaging of the build
//Exemple aar or apk or jar"
POM_PACKAGING = "apk"

//Define How we can retrieve the source of the project
//-----
//Have a look here to understand this section
//https://maven.apache.org/scm/git.html

//Give us a description of the project (optional)
POM_DESCRIPTION = "This is the apk of the Basile Multiplication"
//The Url where your project lie (only for information)
POM_URL = "https://github.com/MathiasSeguy-Android2EE/MultiplicationBasile"
//The url to connect to push/pull source
POM_SCM_URL = "https://github.com/MathiasSeguy-Android2EE/MultiplicationBasile"
//The connection of the source
POM_SCM_CONNECTION = "scm: git:ssh://github.com/MathiasSeguy-Android2EE/MultiplicationBasile"
//The connection of the source as a developper
POM_SCM_DEV_CONNECTION = "scm: git:ssh://github.com/MathiasSeguy-Android2EE/MultiplicationBasile"
//Year of the birth of the project
POM_INCEPTION_YEAR = "2017"

//Define your licenec type
//-----
POM_LICENCE_NAME = "Apache2"
POM_LICENCE_URL = "https://www.apache.org/licenses/LICENSE-2.0"
POM_LICENCE_DIST =

//Define your name
//-----
POM_DEVELOPER_ID = "MSE_A2EE"
POM_DEVELOPER_NAME = "Seguy Mathias"
POM_DEVELOPER_MAIL = "mathias.seguy@android2ee.com"
POM_DEVELOPER_ROLE = "Android Expert"

//Define where to deploy
//-----
NEXUS_USERNAME = "Name"
NEXUS_PASSWORD = 'Password'
NEXUS_URL = "http://nexus.somewhere.com:8081"

SNAPSHOT_REPOSITORY_URL =
"http://nexus.somewhere.com:8081/repository/basile_multiplication_apk_snap"
RELEASE_REPOSITORY_URL =
"http://nexus.somewhere.com:8081/repository/basile_multiplication_apk"

//Define your GPG for signing your archives
//-----

```

```

//signing.keyId=
//signing.password=
//signing.secretKeyRingFile=

//Reference:
//http://zserge.com/blog/gradle-maven-publish.html
//https://github.com/A-CMS/alliance-platform-parent/blob/master/pom.xml
//https://maven.apache.org/scm/git.html
}

```

We will use them in our script to upload our archives. We will override the uploadArchives task to upload the default apk on Nexus.

You have 3 main steps:

- Define in the mavenDeployer section, the coordinates of the project
- and its pom project description
- You can after that add tasks (source, Javadoc...) using the artifact pattern

We use and reuse the variables we have defined in nexus_apk_var_def.gradle file.

```

uploadArchives {
    group = project.ext.myGradleGroup
    description = "Upload the achive in Nexus repository. The selected archive is defined in build.gradle" +
        "in the android."
    repositories {
        mavenDeployer {
            //Define the Coordinates of the project
            //-----
            pom.artifactId = project.POM_ARTIFACT_ID
            pom.groupId = project.POM_GROUP_ID
            //If you are on master, push on release
            if (project.def_branche_name.contains('master')
                ||project.def_branche_name.contains('dev')){
                println 'ReleaseVersion'
                pom.version = project.VERSION_NAME
                repository(url: project.RELEASE_REPOSITORY_URL) {
                    authentication(userName: project.NEXUS_USERNAME, password:
project.NEXUS_PASSWORD)
                }
            } else {
                println 'Snapshot'
                pom.version = project.VERSION_NAME+'-SNAPSHOT'
                repository(url: project.SNAPSHOT_REPOSITORY_URL) {
                    authentication(userName: project.NEXUS_USERNAME, password:
project.NEXUS_PASSWORD)
                }
            }
        }

        //Describe your POM and its content
        pom.project {
            name project.POM_NAME
            packaging project.POM_PACKAGING
            description project.POM_DESCRIPTION+'for the apk '+project.def_apk_name_wrelease
            url project.POM_URL

            scm {
                url project.POM_SCM_URL
                connection project.POM_SCM_CONNECTION
            }
        }
    }
}

```

```

        developerConnection project.POM_SCM_DEV_CONNECTION
    }

    licenses {
        license {
            name project.POM_LICENCE_NAME
            url project.POM_LICENCE_URL
            distribution project.POM_LICENCE_DIST
        }
    }

    developers {
        developer {
            id project.POM_DEVELOPER_ID
            name project.POM_DEVELOPER_NAME
        }
    }
}

}

}

//Add the Javadocs, the source jar tasks and some more if you want
task androidJavadocsJar(type: Jar, dependsOn: generateProjectJavadocs) {
    classifier = 'javadoc'
    from generateProjectJavadocs.destinationDir
}

task androidSourcesJar(type: Jar) {
    classifier = 'sources'
    from android.sourceSets.main.java.sourceFiles
}
//Add those tasks as artifacts to have them took into account
artifacts {
    archives androidSourcesJar
    archives androidJavadocsJar
}

//You could have had signing and javadocs also.
// signing {
//     required { isReleaseBuild() && gradle.taskGraph.hasTask("uploadArchives") }
//     sign configurations.archives
// }

// task androidJavadocs(type: Javadoc) {
//     failOnError false // add this line
//     source = 'src/main/java'
//     source = android.sourceSets.main.java.getSrcDirs()
// }
}

```

And it should works.

25 Taking care of libraries

As a developer, you should respect the open source libraries much more than your managers. And a good way to do that is to display the reward associated with the libraries you use. And by the way, you can be suit if you don't respect the OSP's licensing of the OSP you use.

For that I use the plugin Android License Tools Plugin (<https://github.com/cookpad/license-tools-plugin>) which does that really well. Have a look at its ReadMe and five minutes later, it's implemented on the project.

In your application build.gradle, link to the plugin

```
buildscript {
    repositories {
        mavenCentral()
        jcenter()
        maven { url 'https://maven.fabric.io/public' }
        google()
    }
    dependencies {
        //...
        //Gradle License Plugin
        classpath 'com.cookpad.android.licensetools:license-tools-plugin:0.23.0'
    }
}
```

Then in the gradle.build of your module (of your application), apply the plugin:

```
/**
 * This plugin provides a task to generate a HTML license report based on the configuration
 * You can find the plugin project here :https://github.com/cookpad/license-tools-plugin
 */
apply plugin: 'com.cookpad.android.licensetools'
```

Now you just have to call the task that generates your licences html file (or json) you can use to display it in a webview in your application. The best is to do it when making a release build. So just make your releaseBuild task depends on it. As simple as that:

```
//Make a full build :clean/build/InstrumentationTests
task releaseBuild(dependsOn: [
    'generateLicensePage',
    // 'enableCrashlytics',
    'buildAndCheckProject'
]) {
```

It's just perfect.

26 Conclusion

Having this tooling on any Android project is bless. I hope you will add them, I hope those articles helped you. Thanks for reading and have a good life.

La bise.