antonioleiva
Become a Kotlin expert in Android

Kotlin DSL to write Gradle scripts on Android: Step by step walkthrough

by Antonio Leiva | Blog, Kotlin | 2 comments

KOTLIN DSL TO WRITE GRADLE SCRIPTS ON ANDROID: STEP BY STEP WALKTHROUGH
ANTONIO LEIVA
antonioleiva.com

There's been quite some months already since Gradle announced that they were working on supporting Kotlin to write Gradle Scripts, by using a version of the language that has been recently revamped to Kotlin DSL.

At the beginning things where quite complicated, but nowadays, with latest versions of Kotlin DSL (at the time of writing this the version is 0.12) the idea is more mature.

So, knowing how, it's not too difficult to start using Kotlin to build your Gradle files in Android.

One of the main issues of this is the lack of documentation, so I decided to write about my experience converting the Gradle files of Bandhook-Kotlin, so that you can replicate it in your project.

If you're still starting with Kotlin, you may be interested in checking out my previous articles about Kotlin.

Using Kotlin DSL on your Gradle files. Is it worth it?

I guess this is the first question to solve. As of today, should I spend my time converting my files to Kotlin DSL?

My answer is probably a bit counterproductive to encourage you continue reading this article, but I don't want you to be hyped because of this: **you probably shouldn't**.

It has of course some pros:

- You can used a language you're more familiarized with, so **it's easier to start doing more complicated things**. I had never done anything on `buildSrc` folder, and it was quite easy for me to create my own class and use it in the rest of the script files.
- The IDE helps you a lot more: **nice autocomplete, the errors are detected by the compiler, imports added automatically**… All you know and love from your regular Kotlin code is kind of extrapolated here.

But also has some cons:

- There's **not a straightforward way to convert from Groovy to Kotlin files**. I'll explain you how to make it easier though
- **You need to know how the plugin is implemented to be able to use it**: where in Groovy you just use an equals to assign a value to every configuration, here you need to know whether it's a function or a property to know how to set it. Gladly the IDE can help. Gradle team says that this will only be solved when people write the plugins thinking also on Kotlin DSL. There are some rules to follow. An example (we'll see more later):

```
applicationId = config.android.applicationId
minSdkVersion(config.android.minSdkVersion)
targetSdkVersion(config.android.targetSdkVersion)
versionCode = config.android.versionCode
versionName = config.android.versionName
```

- **There's not much documentation** or examples: I think this is the main problem. If you get stuck, it's difficult to continue. It took me quite some time (and asking on the great Kotlin Slack) to know how to do some things.

So a big disclaimer here: **be sure of what you're doing if you use it in your production code**. Can be a interesting thing to do in your pet projects though. I'm not saying it's not mature enough, just that it's not easy to use.

How to convert your files

I want to give you here the fastest route, by skipping all the pain points I had to solve. So if I had to convert another project to use Kotlin on Gradle files, that's what I'd do.

Use the latest version of Gradle

The newer the Gradle version, the better, because it will include the latest Kotlin DSL version. When I converted this project, the latest one was 4.3. You can check the latest release here. Modify your `gradle-wrapper.properties` file to use it:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/
```

Change the name of your files

I must admit that, since first time I tried, things have improved a lot. Before, you had to add many configurations that you wouldn't need when using Groovy.

Now, you just need to add an extension to your `build.gradle` files, and Gradle will be able to use Kotlin files. You just need to rename them to `build.gradle.kts`.

Compile using the terminal

The IDE won't help you much here in understanding what's wron, so I recommend you using *cmd* and the `--info` flag:

```
./gradlew assembleDebug --info
```

With this, you'll get a better idea of what's not working. You can do it now if you want, but it will obviously fail.

Configure your buildSrc folder

One of the pain points I found was a way to replicate the `ext` object in Groovy, that allows you to share variables between Groovy files:

```
ext {
    // Android config
    androidBuildToolsVersion = "26.0.2"
    ...
}
```

You can have that in your root Gradle file, and then use it in your modules:

```
buildToolsVersion parent.ext.androidBuildToolsVersion
```

That's pretty easy, and works fine. The alternative in Kotlin DSL is this: for each variable you need to create an extra like this:

```
var androidBuildToolsVersion: String by extra
androidBuildToolsVersion = "26.0.2"
```

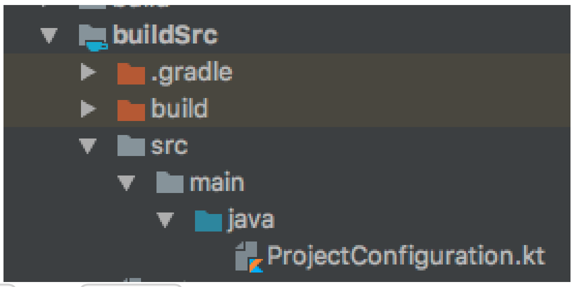
And then you use it with this:

```
val androidBuildToolsVersion: String by extra
buildToolsVersion(androidBuildToolsVersion)
```

As you can imagine, this doesn't scale up very well. Having all this code for each variable is a pain.

So the alternative I found is to create a configuration file in the `buildSrc`, and add all you need in an object that you can then instantiate in any Gradle files. If you don't know about it, the `buildSrc` is basically a place where you put all the code that you want to use while building the project scripts. You can find more info in Gradle Docs.

To configure it, just create this folder structure under the `buildSrc` folder:



Forget about `.gradle` and `build` folders and create the rest.

Under that folder, also create a new `build.gradle.kts` with this content:

```
plugins {
    `kotlin-dsl`
}
```

The `ProjectConfiguration` file will be the one to hold the variables you may use in your project. You can use whatever organization you want. While looking for info on how to do this, I found the repository from Arturo Gutiérrez that uses this structure, and I liked it:

```
class ProjectConfiguration {
    val buildPlugins = BuildPlugins()
    val android = Android()
    val libs = Libs()
    val testLibs = TestLibs()
}
```

Then, each child object has its own values. For instance, the *Android* one:

```
class Android {
    val buildToolsVersion = "26.0.2"
    val minSdkVersion = 19
    val targetSdkVersion = 26
    val compileSdkVersion = 26
    val applicationId = "com.antonioleiva.bandhookkotlin"
    val versionCode = 1
    val versionName = "0.1"
}
```

You can also use some top variables to make it easier to edit:

```
val supportVersion = "26.1.0"
...
class Libs {
    val appcompat = "com.android.support:appcompat-v7:$supp
    val recyclerview = "com.android.support:recyclerview-v
    val cardview = "com.android.support:cardview-v7:$suppo
    val palette = "com.android.support:palette-v7:$support
    val design = "com.android.support:design:$supportVersi
    ...
}
```

Then, you will be able use that in your `build.gradle` files:

```
val config = ProjectConfiguration()
...
    defaultConfig {
        applicationId = config.android.applicationId
        minSdkVersion(config.android.minSdkVersion)
        targetSdkVersion(config.android.targetSdkVersion)
        versionCode = config.android.versionCode
        versionName = config.android.versionName

        testInstrumentationRunner = "android.support.test.
    }
```

As you see, it looks much cleaner.

Keep converting your Gradle files

Until the first time it completely compiles, you'll have to rely on what the terminal builds say, the IDE won't be very useful here.

So continue changing parts little by little, building the project, and interpreting the output. As a reference, I can leave you some parts of the Gradle files here (and you can, of course, check the complete project on Github).

For the root `build.gradle`:

```
buildscript {

    val config = ProjectConfiguration()

    repositories {
        jcenter()
        google()
    }
    dependencies {
        classpath(config.buildPlugins.androidGradle)
        classpath(config.buildPlugins.kotlinGradlePlugin)
    }
}

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

This one becomes quite simple, as we've extracted all kinds of configuration to the `ProjectConfiguration.kt` file.

The module file is a bit more complicated. For the plugins, you do it like this:

```
plugins {
    id("com.android.application")
    kotlin("android")
    kotlin("kapt")
}
```

For regular plugins, you just use the function `id`, and Kotlin plugins use the function `kotlin`.

Then the Android section is like you saw above. You need to try in order to discover whether it's a function or a property. Check the repository the most typical ones. Then, for the build types:

```
buildTypes {
    getByName("release") {
        isMinifyEnabled = false
        proguardFiles("proguard-rules.pro")
    }
}
```

You can't just create a `release` block, but instead it's required to find it by name, and then you can configure it.

The dependencies are easy, just functions where you set the name of the dependency:

```
dependencies {
    compile(config.libs.kotlin_std)
    ...
}
```

Keep building and polishing until the build succeeds.

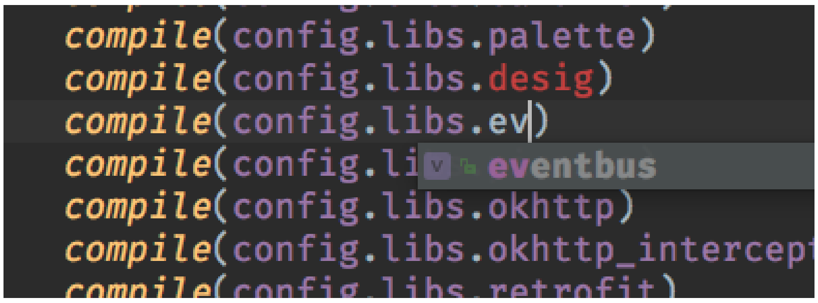
It's not easy, but it's cool!

It's really awesome to see how Kotlin is reaching to all development environments: JVM, JS, Gradle... and potentially everywhere with Kotlin/Native.

This is just another example of the versatility of the language, and gives an idea of how enthusiastic the different developers communities are becoming about it.

In the case of Gradle, maybe it's not yet production ready (though I know of people that are using it without many issues), but **it's worth giving it a try and check how nice it works** once everything is configured.

Having **compile time errors and autocomplete** is of great help when we're building our Gradle files, which otherwise requires just hard memory or searching.



```
compile(config.libs.palette)
compile(config.libs.desig)
compile(config.libs.ev)
compile(config.li
compile(config.libs.okhttp)
compile(config.libs.okhttp_intercept
compile(config.libs.retrofit)
```

What do you think about Kotlin DSL? Have you tried? Are you using it in your projects? **Let me know in the comments.**

Author: Antonio Leiva

I'm in love with Kotlin. I've been learning about it for a couple of years, applying it to Android and digesting all this knowledge so that you can learn it with no effort.

Twitter Google+ Linkedin Github

Share this:

More

Like this:

★ Like

Be the first to like this.

Related

12 REASONS WHY YOU SHOULD START USING KOTLIN FOR ANDROID TODAY
KAD 30
ANTONIO LEIVA
antonioleiva.com

12 reasons why you should start using Kotlin for Android today (KAD 30)

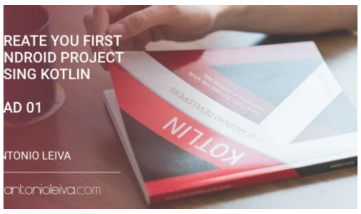
July 11, 2017

In "Blog"

Kotlin for Android (II): Create a new project

March 23, 2015

In "Blog"

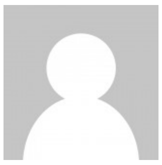


Create your first Android project using Kotlin (KAD 01)

November 21, 2016

In "Blog"

2 Comments



Pepa Hruška (@M3llOun) on November 24, 2017 at 01:48

Cool article. I just started converting gradle to kotlin. I noticed that you are modifing gradle.properties directly. Maybe it is easier to just run this gradle task: ./gradlew wrapper –gradle-version=4.4-rc-1

What do you think?

Reply

Antonio Leiva on November 24, 2017 at 08:32

Have never used it honestly. Found out that this exists very recently. But yeah, guess it's a little easier. Thanks for sharing!

Reply