Migrating from Gradle to Gradle

Gradle build scripts have been written in a Groovy based DSL for a long time. Although flexible, the IDE support was always a bit of a problem. You either knew what to type or you searched the docs or tried to find an answer on stackoverflow. IDEs always struggled to provide help on writing tasks or configuring them.

For some time now, a Kotlin based DSL is in the works and as of **Gradle 5** it is available in 1.0. So is it any better compared to what you can do with the Groovy based DSL?

To get started, some reading the documentation (later, after this blog post!) helps:

- Gradle Kotlin DSL Primer
  This is an interesting read, but for me it was also a hard one as it goes very into gradle internals
- Migrating build logic from Groovy to Kotlin
  if you have builds with a lot of custom tasks or write your own gradle plugins don't miss the section on configuration-avoidance
- Kotlin DSL Samples
  Have a look and some example builds

If you need to learn about Gradle in general, there are free online trainings available that I can highly recommend (from starting with gradle to advanced topics).

The **example project created for this comparison is on GitHub** and contains a simple spring boot application also written in kotlin that spits out a docker image. The master branch uses the groovy DSL, the kts branch uses the new kotlin DSL but does exactly the same.

Overview of the groovy build

The groovy based build script uses the new plugin syntax:

new plugin syntax

```
plugins {
  id "com.palantir.docker" version "0.22.1"
}
```

Instead of the old syntax which would look like this:

old plugin syntax

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "gradle.plugin.com.palantir.gradle.docker:gradle-docker:0.22.1"
  }
}

apply plugin: "com.palantir.docker"
```

This will simplify the kotlin script migration, as the kotlin syntax is very similar to the new one.

Note on the new plugin syntax

There have been some issues with this new syntax when a Maven Repository Proxy (like Nexus or Artifactory) is used. But the Gradle plugin repository is available as maven repository as well and as of Gradle 4.4.x plugins can be loaded via a repository proxy too (previously this would only work without any authentication or with direct internet access - which is unlikely in an enterprise environment). So Gradle 4.4.x comes to the rescue! You can add your repository proxy to an init.d script and use the new plugin syntax.

$HOME/.gradle/init.d/repository.gradle

```
apply plugin: EnterpriseRepositoryPlugin

import org.gradle.util.GradleVersion

class EnterpriseRepositoryPlugin implements Plugin<Gradle> {

    private static String NEXUS_PUBLIC_URL = "https://<nexushostname.domain>/repository/public"

    void apply(Gradle gradle) {
        gradle.allprojects { project ->
            project.repositories {
                maven {
                    name "NexusPublic"
                    url NEXUS_PUBLIC_URL
                    credentials {
                       def env = System.getenv()
                       username "$env.NEXUS_USERNAME"
                       password "$env.NEXUS_PASSWORD"
                    }
                }
            }

            project.buildscript.repositories {
                maven {
                    name "NexusPublic"
                    url NEXUS_PUBLIC_URL
                    credentials {
                        def env = System.getenv()
                        username "$env.NEXUS_USERNAME"
                        password "$env.NEXUS_PASSWORD"
                    }
                }
            }
        }

        def referenceVersion = GradleVersion.version("4.4.1")
        def currentVersion = GradleVersion.current();
        if(currentVersion >= referenceVersion) {
            gradle.settingsEvaluated { settings ->
                settings.pluginManagement {
                    repositories {
                        maven {
                            url NEXUS_PUBLIC_URL
                            name "NexusPublic"
                            credentials {
                                def env = System.getenv()
                                username "$env.NEXUS_USERNAME"
                                password "$env.NEXUS_PASSWORD"
                            }
                        }
                    }
                }
            }
        } else {
            println "Gradle version is too low! UPGRADE REQUIRED! (below " + referenceVersion + "): " + gradle.
gradleVersion
        }
    }
}
```

Other than that the build does not contain any unusual things. There are task configurations and a custom task. The spring boot dependencies are added, the test task is configured to measure the test coverage using jacoco. The build uses the docker-palantir plugin to create the docker image. And there is a task that prints some log statements to tell a TeamCity server about the coverage results (will not hurt on Jenkins or Bamboo).

The docker plugin uses task rules to create some tasks, so it's configured via the extension class - there are several ways to do this, the variant used in the example is also to have IntelliJ understand what task it is so auto-completion works.

Overview of the kotlin build

The kotlin DSL build script (in `build.gradle.kts` - the buildscript has a new filename!) uses a plugin syntax very similar to the groovy one. It looks almost the same. The build script looks very similar if you compare them both.

The way tasks are referenced or created changes slightly. Once you get used to it it's fairly easy to use, and the IDE supports what you are doing!

Quirks and conclusion

As the IDE support is currently limited to IntelliJ we can only look at that. But if you were used to compile gradle builds on the command line anyway, the gradle wrapper will automatically recognize the kotlin build script and is by default capable to run these.

An improvement you almost immediately notice: auto-completion in build scripts suddenly makes sense! For some reason it sometimes is very slow to show up but the suggestions made are way better than what you would see using the groovy builds.

Yet IntelliJ will sometimes mark fields of plugins as not accessible - the build will work, it's just the IDE that complains. There are workarounds for some of the warnings.

The expected variant:

```
tasks.test {
    extensions.configure(JacocoTaskExtension::class.java) {
        destinationFile = file(jacocoExecFileName)
    }
}
```

But there is some access warning. But you can switch to using the setter:

```
tasks.test {
    extensions.configure(JacocoTaskExtension::class.java) {
        setDestinationFile(jacocoExecFileName)
    }
}
```

Not too nice, but not a showstopper - and unclear if Gradle is to blame or if it's some IntelliJ issue.

In other cases it helped to hint the task type:

```
tasks.withType(Test::class.java) {
}
```

To have better auto-completion. The documentation on the kotlin DSL gives some hints how to help yourself.

In every case where IntelliJ complained, a workaround could be found. But these are just to have IntelliJ not complain on the build script. Not ideal. But you can reach a state where IntelliJ does not mark any line with an error or warning! Compared to the random errors and warnings mess in the groovy build scripts: **way better.**

Comparing the length of both scripts, doesn't really show a clear winner. Both scripts have about the same length and structure. The tasks are often a few lines shorter, but the type declarations will add an import statement on top. Overall this simplifies the migration and keeps the readability one got used to. I wished everything was just shorter and more expressive - but that probably was just a personal wish - actually it is a bit unfair to the groovy DSL, which is already good. The build scripts seem to initialize slower but builds run at the same speed. But the way gradle optimizes task execution or determines if task configuration needs to be loaded at all did improves with gradle 5 - so the speed penalty might not be there for you at all. So the way it looks today: quite good :-)

I have no concerns to use the kotlin DSL in production builds at all and the IntelliJ support is in a good state, so you will not need to flip between the IDE and the command line all the time if you don't like doing that.

Is it a migration?

The title states this was a Gradle to Gradle migration. But the resulting build scripts look very similar. So is it really one? I would say yes. It took me two attempts with a couple of hours of searching around in the documentation and experimenting (as there are not so many examples around yet). Although the result does not look like much of a change, it took some effort to get there. But effort in the meaning of hours to days - surely not weeks (as I'm not the most experienced gradle nor kotlin user). Of course this may fall apart if a lot of plugins are used or they don't properly interact with the Gradle API in this version (as you will probably upgrade to gradle 5.x from a 4.x version).

Hints for the hasty

The linked documentation on top already contains this but just in case you are a very hasty developer, here are some useful gradle tasks in this context:

```
$ gradle kotlinDslAccessorsReport
```

prints the Kotlin code necessary to access the model elements contributed by all the applied plugins. The report provides both names and types.

You can then find out the type of a given task by running

```
$ gradle help --task <taskName>
```

Another important statement is in the migration guide in the configuring plugins section:

> **Keeping build scripts declarative**
>
> To get the most benefits of the Gradle Kotlin DSL you should strive to keep your build scripts declarative. The main thing to remember here is that in order to get type-safe accessors, plugins must be applied before the body of build scripts.

So if you are programming in kotlin anyway, and you also use TeamCity and its kotlin build DSL you can now also use kotlin in your builds too. Kotlin all the things!

Kotlin has certainly more momentum today compared to Groovy. The typed DSL solves some crucial handling issues in gradle build scripts. I would guess the new DSL may become the default at some point, not that I'm aware of any timeline. Just an assumption. So don't hurry, the groovy DSL will be around for quite some time. If you are starting with gradle, I would try using the kotlin DSL from the beginning.

Give it a try!