RUN KOTLIN SCRIPTS (KTS) FROM REGULAR KOTLIN PROGRAMS

12 MAY 2018

3 COMMENTS

Run Kotlin Scripts from Kotlin Programs

This article presents a way to run Kotlin scripts from Kotlin programs in order to leverage the power of DSLs.

Kotlin can be used as a scripting language. Simply write top-level executable code inside a file with `.kts` extension and run it with the `kotlinc` as described in the documentation. That's also the format of Gradle build files that are used in combination with the Gradle Kotlin DSL like this gradle.build.kts. Gradle shows a fantastic example of a domain specific language that can be written standalone in `.kts` files to be read by the `gradle` tool later on. When we try to find a way to do the same with custom DSLs (Tutorial can be found here), we first need to know how to run Kotlin scripts from Kotlin programs. The article reveals how to do so.

The Java Scripting API (JSR-223)

The Java Scripting API is a tool for using scripting engines (such as Nashorn) from Java code. It enables users to write customizable scripting code that can be picked up by the Java application at runtime. In a way, the API is a neat way of writing extensible applications.

As of Kotlin 1.1, the corresponding JSR-223 is supported for Kotlin Scripts, too. That means that it's possible to run Kotlin scripts from regular Kotlin programs in order to make applications customizable through these scripts.

Using the Kotlin Script Engine

In order to use the mentioned Kotlin script engine, a file called `javax.script.ScriptEngineFactory` has to be placed inside `META-INF/services` of your application. It should contain the following entry: `org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory`.
After that, the `javax.script.ScriptEngineManager` will be able to find the corresponding engine when looked up via `ScriptEngineManager().getEngineByExtension("kts")`. This code now finds the Kotlin `ScriptEngine` implementation, an instance that can be used to evaluate `String`-based scripts such as `"5 + 2"`, or directly read scripts from the file system. Here's a short example:

```
with(ScriptEngineManager().getEngineByExtension("kts")) {
    eval("val x = 3")
    val res2 = eval("x + 2")
    assertEquals(5, res2)
}
```

You could also compile scripts and evaluate them later:

```
val script = compile("""listOf(1,2,3).joinToString(":")""")
assertEquals(listOf(1, 2, 3).joinToString(":"), script.eval())
```

Wrapping the glue code in a library

Twitter

Simon Wirtz Follow

s1m0n Simon Wirtz @s1m0nw1 · 12h
You should watch this. Márton is super enthusiastic and knowledgable about #kotlin and you will learn a lot from this talk

AutSoft @autsoftltd
Our recent article, Delightful

Recent Posts

Kotlin Inline Classes – How they work and when you should use them

How Kotlin makes me a more productive software developer

Concurrent Coroutines – Concurrency is not Parallelism

Hibernate with Kotlin – powered by Spring Boot

Execute Kotlin Scripts with Gradle

the very first version, provides a slim `KtsObjectLoader` class whose usage is shown in the following example:

```
data class ClassFromScript(val x: String)


import de.swirtz.ktsobjectloader.ClassFromScript

ClassFromScript("I was created in kts")
```

The previous snippets show the definition of some arbitrary data class and the code that instantiates an object of it. The object instantiation is basically what we write into a `.kts` file.

```
val scriptReader = Files.newBufferedReader(Paths.get("path/classDeclaration.kts"))
val loadedObj: ClassFromScript = KtsObjectLoader().load<ClassFromScript>(scriptReader)
assertEquals("I was created in kts", loadedObj.x)
```

Using the `KtsObjectLoader` makes it simple to load the correspoding object of `ClassFromScript` from the script file. Alternatively, the script could also be provided as a `String`:

```
val scriptContent = "5 + 10"
val result: Int = KtsObjectLoader().load<Int>(scriptContent))
assertEquals(15, result)
```

Adequate Usage Scenario

As mentioned in the beginning, it can make sense to make your application customizable through external scripts, similar to how Gradle can be extended with any custom build script. Imagine an application that provides a test suite runtime. The actual test cases are provided by technical testers who write their test scripts using a domain specific language that is provided by the main application. Since you don't want testers to add source files (defining new test cases) to your application all the time, the test case creation is made in independent `.kts` files in which the DSL is utilized by the testing team. The test suite main application can use the presented KtsRunner library for loading the test cases provided in `.kts` files and process them further afterward.

An example

A pretty popular DSL for Kotlin is kotlinx.html, a language for describing type-safe HTML. You let the client of your application provide some arbitrary HTML that you want to render at a later time. The HTML DSL code is provided as `.kts` script files and might look like this:

```
import kotlinx.html.*
import kotlinx.html.dom.create
import org.w3c.dom.Element
import java.io.OutputStream
import java.io.OutputStreamWriter
import javax.xml.parsers.DocumentBuilderFactory

val document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument()
document.create.html {
    head {
        title("Hello world")
    }
```

```
                +"paragraph1"
            }
        }
    }
```

When executed, an instance of `org.w3c.dom.Element` is created that contains the described HTML code in an XML document:

```
<?xml version="1.0" encoding="UTF-8"?><html>
    <head>
        <title>Hello world</title>
    </head>
    <body>
        <h1 class="h1Class" style="background-color:red">My header1</h1>
        <p class="pClass">paragraph1</p>
    </body>
</html>
```

That's straightforward but the interesting part is that the script should actually be executed from the main program. For this purpose, we add the KtsRunner to the application by adding a repository and the dependency itself to the Gradle build file:

```
maven {
    setUrl("https://dl.bintray.com/s1m0nw1/KtsRunner")
}


dependencies {
    //...
    compile("de.swirtz:ktsRunner:0.0.x")
}
```

The final code for loading the `Element` from the external script looks as follows:

```
KtsObjectLoader().load<Element>(script)
```

Simple, isn't it? Unfortunately, the shown Scripting API implementation for Kotlin is rather slow and you'll definitely notice some performance constraints. Altogether, the KtsRunner is a very tiny tool that only encapsulates the glue code for enabling Kotlin Scripting support in random applications. The library is published on bintray and can therefore easily be used from your own application.

Please follow and like this Blog 🙂

s1m0nw1

Simon is a software engineer based in Germany with 7 years of experience writing code for the JVM and also with JavaScript. He's very passionate about learning new things as often as possible and a self-appointed Kotlin enthusiast.

%2Finformation.za.org%2Fprofile.php%3Fa%3D10692%3ERapid+Tone%3C%2Fa%3E

May 14, 2018 at 4:12 PM Reply

Pabl0rg

That's pretty cool. It makes sense that script compilation can be slow, once the script is compiled and you've loaded the Element (in the khtml example), is use of that object slower than the use of any other?

May 20, 2018 at 3:27 PM Reply

Kane

How to not explicitly importing custom dsl object like build.gradle.kts? say as `dependencies` in build.gradle.kts

November 22, 2018 at 10:44 AM Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name* Email* Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

Post Comment

SOCIAL