Authoring Gradle Builds with the Kotlin DSL

KOTLIN/Everywhere Cologne 2019

Gradle

About me

Benedikt Ritter - Senior Software Engineer at Gradle Inc.

⍰⍰ @BenediktRitter

🐱 @britter

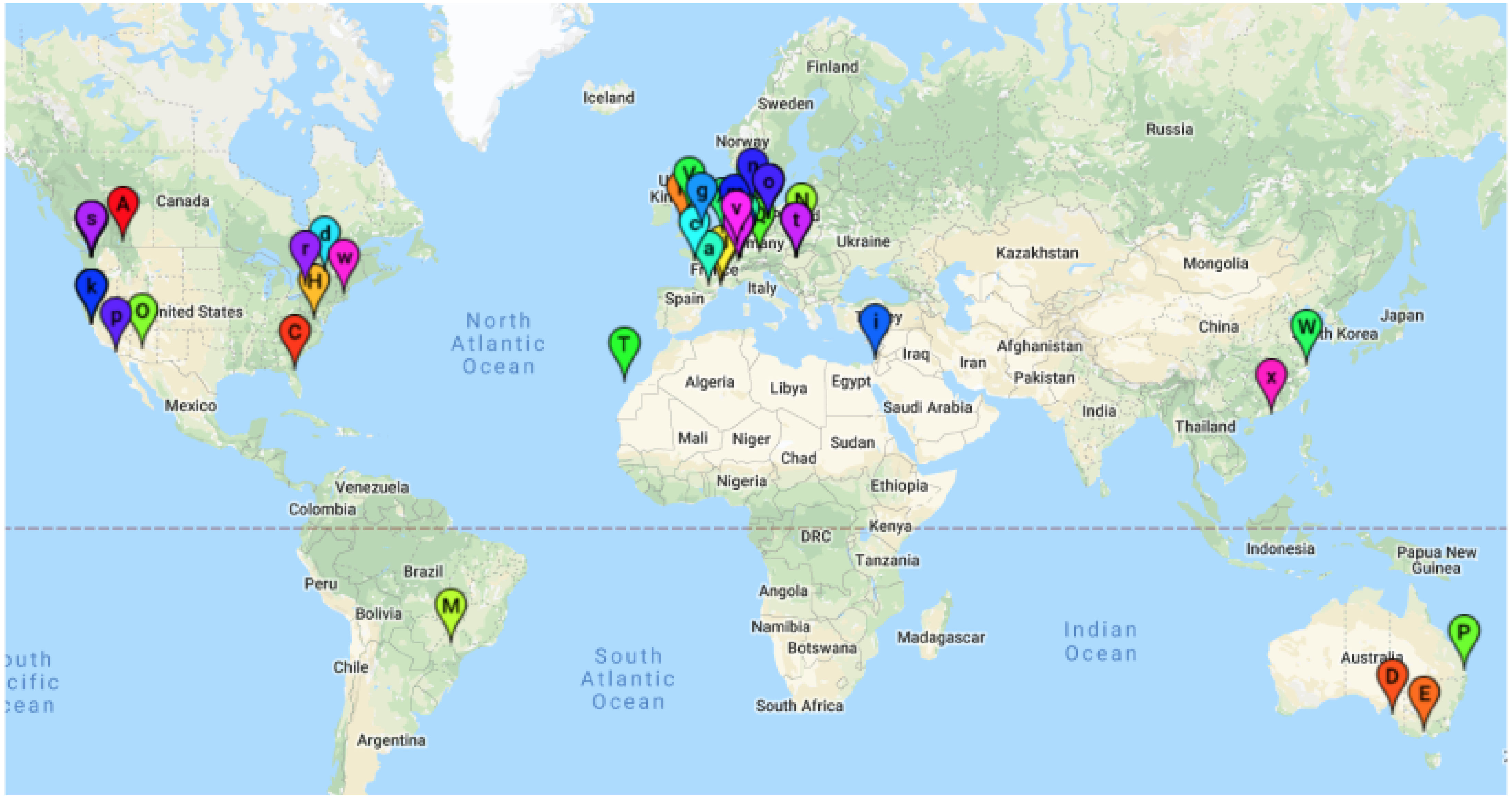
🎙 autoweird.fm

Agenda

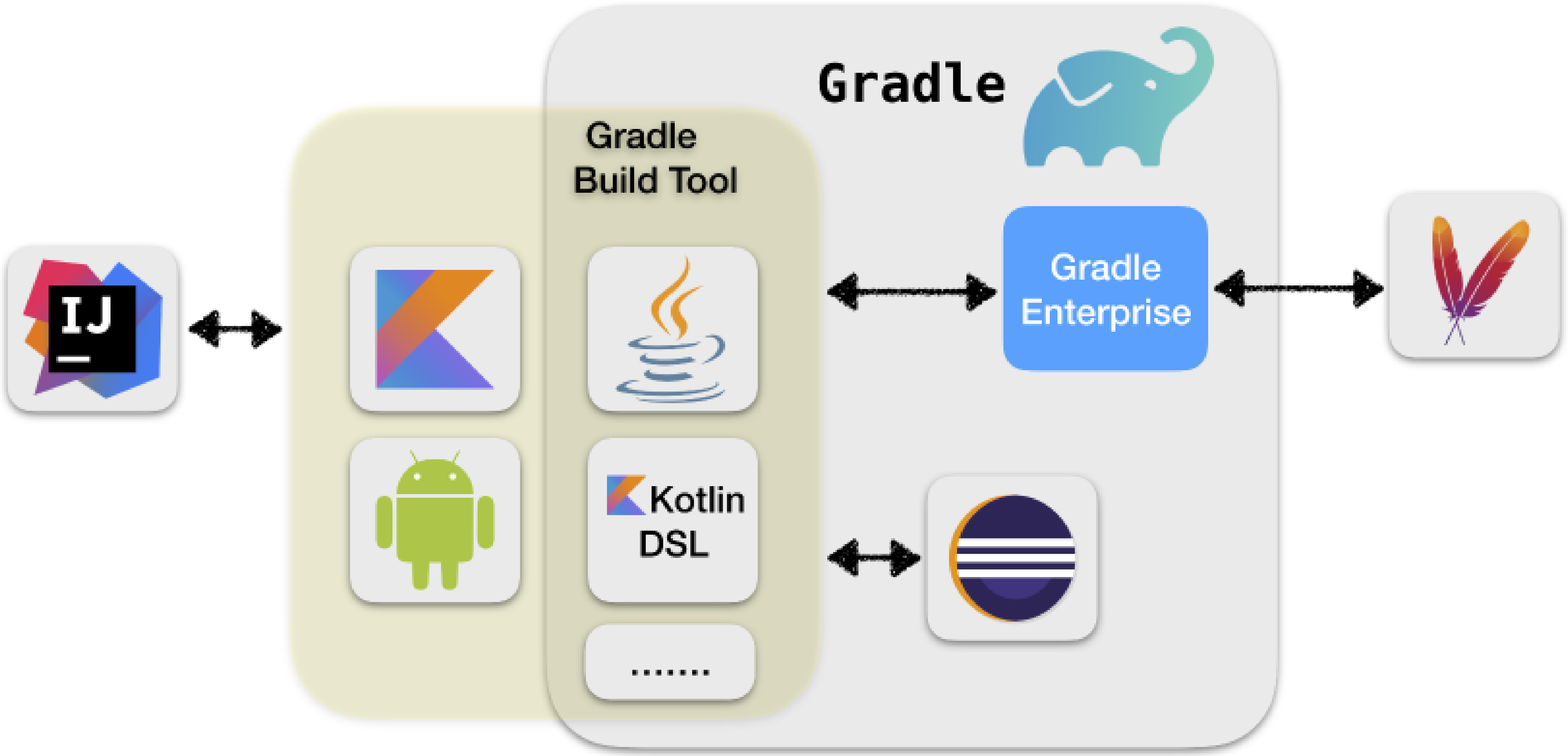
- Gradle Introduction
- What powers the Gradle Kotlin DSL
- Pre-compiled Script Plugins
- Tipps for authoring Plugins

Gradle Introduction

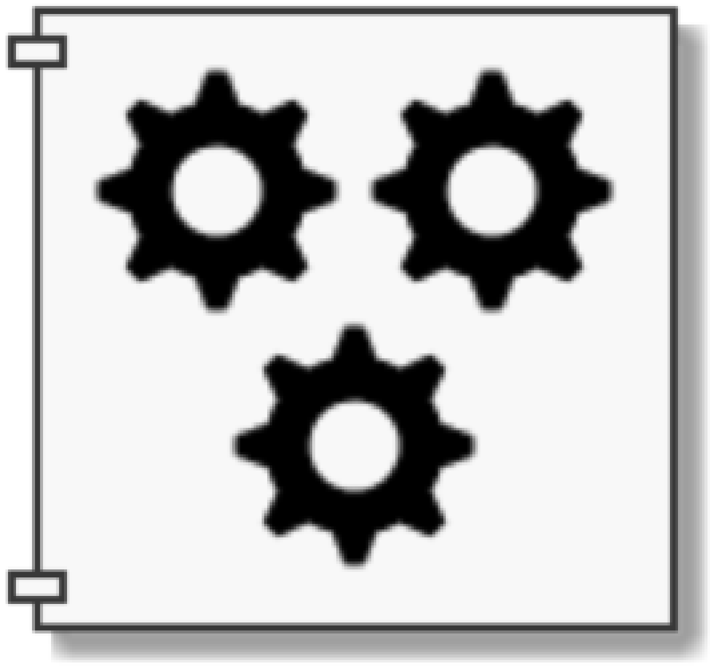
Who is Gradle Inc.?



What is Gradle?



Your Project as a Software Component



You define a model of your software project through a Java/Kotlin API

Entry point: `org.gradle.api.Project`

Authoring a Build

`build.gradle.kts`

```kotlin
// empty file
```

`settings.gradle.kts`

```kotlin
rootProject.name = "kotlin-everywhere"
```

Authoring a Build

build.gradle.kts

```
tasks.register("t1") {
    inputs.dir("...")
    ouputs.file("...")
    doLast {
        // do some work
    }
}
```

settings.gradle.kts

```
rootProject.name = "kotlin-everywhere"
```

Extending the Gradle DSL

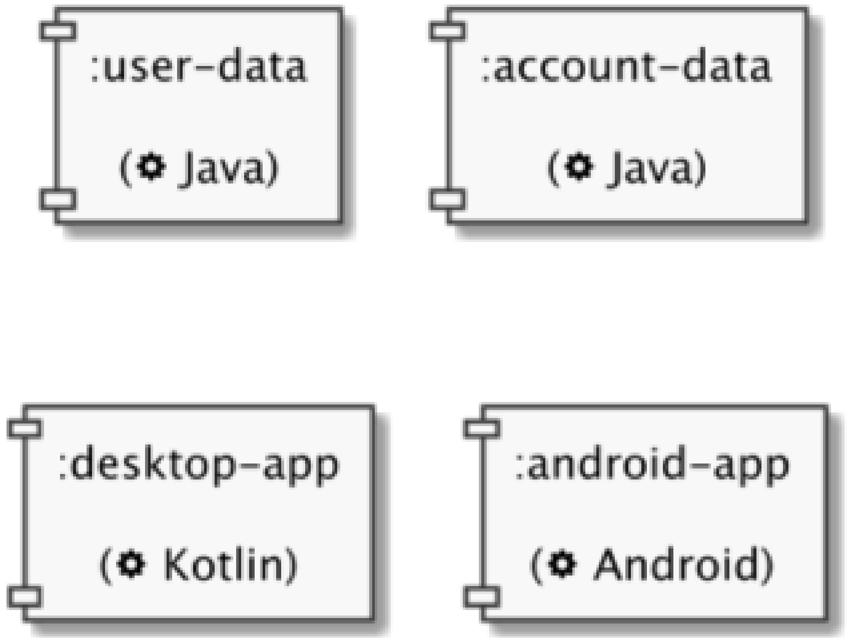
`buildSrc/src/main/kotlin/MyProject.kt`

```
fun org.gradle.api.Project.configureMyProject() {
    tasks.register("t1") {
        inputs.dir("...")
        ouptuts.file("...")
        doLast {
            // do some work
        }
    }
}
```

`build.gradle.kts`

```
configureMyProject()
```

Your Project as multiple Software Components



build.gradle.kts

```
project(":user-data").plugins.apply("java")
project(":account-data").plugins.apply("java")
project(":services").plugins.apply("org.jetbrains.kotlin.jvm")
project(":desktop-app").plugins.apply("org.jetbrains.kotlin.jvm")
project(":android-app").plugins.apply("org.jetbrains.kotlin.android")
project(":android-app").plugins.apply("com.android.application")
```

What powers the Kotlin DSL

Gradle ⟷ Kotlin

- Gradle provides a dynamic model that
  - … can be configured by build authors
  - … can be extended by plugin authors
- The Kotlin DSL provides
  - … a statically typed facade on top of that model
  - … support for authoring builds
  - … the complete Gradle feature set

Kotlin idioms in the DSL

- Delegated Properties
- Extension functions
- Implicit receivers

Delegated Properties

Delegated properties are syntactic sugar for accessing values of a certain type. Values can be accessed using the `by` keyword if the containing object implements:

```
fun getValue(thisRef: Any?, property: KProperty<*>): T
```

This works for example for Maps:

```
val map = mapOf("name" to "Benedikt")
val name: String by map
println(name) // prints "Benedikt"
```

Delegated Properties in Gradle

build.gradle.kts

```
val myTask = tasks.register<Copy>("myTask") {
    // configure copy
}
```

vs.

build.gradle.kts

```
val myTask: Copy by tasks.registering {
    // configure copy
}
```

Delegated Properties in Gradle

Works with all `DomainObjectContainers`

`build.gradle.kts`

```
val integTest: SourceSet by sourceSets.creating {
    compileClasspath = files(configurations.compileClasspath)
    runtimeClasspath = files(configurations.runtimeClasspath)
}

val optionalDeps: Configuration by configurations.creating {
    isCanBeResolved = true
    isCanBeConsumed = false
}
```

Extension functions

Extend any class with additional functionality:

`StringExtensions.kt`

```kotlin
fun String.capitalize() {
    // go through this string and make first characters upper case
}
```

`SomeClass.kt`

```kotlin
import capitalize

fun someMethod(str: String)
    str.capitalize()
}
```

Extension functions in Gradle

`build.gradle.kts`

```
plugins {
    java
}

java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks.test {
    useJUnitPlatform()
}
```

Extension functions in Gradle

`build.gradle.kts`

```
plugins {
    java
}

java { // where is this coming from?
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks.test {  // where is test coming from?
    useJUnitPlatform()
}
```

Extension functions in Gradle

Somewhere inside the Kotlin DSL implementation (simplified)

`build.gradle.kts`

```kotlin
val org.gradle.plugin.use.PluginDependenciesSpec.`java`

fun org.gradle.api.Project.java(configure: Action<JavaPluginExtension>)

val org.gradle.api.TaskContainer.`test`: TaskProvider<Test>
```

DSL extensions for plugins that are added to the build are generated on the fly!

Receivers in Kotlin

The receiver of a call is the object the call will executed against.

```
class Guitar {

    fun playTune() { }

    fun playSong(): {
        playTune() // implicit receiver
        this.playTune() // explicit receiver
    }
}
```

Implicit receivers in Gradle

Let’s take a step back and think about build scripts again…

`build.gradle.kts`

```
tasks.test {
    useJUnitPlatform()
}
```

- Where is `tasks` coming from?
- Who is the receiver of the `useJUnitPlatform()` call?

Inside Kotlin build scripts…

- Project is set as an implicit receiver
- All types from the Gradle API are implicitly imported
- Configuration lambdas have an implicit receiver of the type that is configured

`build.gradle.kts`

```
project.tasks.test { test ->
    test.useJUnitPlatform()
}
```

Pre-compiled script plugins

Managing Build configuration

`subproject-a/build.gradle.kts`

```
plugins {
    kotlin("jvm")
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
}

tasks.withType<KotlinCompile> {
    kotlinOptions {
        freeCompilerArgs = listOf("-Xjsr305=strict")
        jvmTarget = "11"
    }
}
```

What happens when we introduce `subproject-b`?

Let’s build a plugin for this!

`buildSrc/src/main/kotlin/KotlinConvetions.kt`

```kotlin
import org.gradle.api.Plugin
import org.gradle.api.Project

open class KotlinConventions: Plugin<Project> {
  override fun apply(project: Project) {
    project.plugins.apply("org.jetbrains.kotlin.jvm:1.3.50")
    project.dependencies.add("implementation",
        "org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.3.50")
    project.tasks.withType(KotlinCompile::class) {
      kotlinOptions {
        freeCompilerArgs = listOf("-Xjsr305=strict")
        jvmTarget = "11"
      }
    }
  }
}
```

Very different developer experience from writing build scripts…

Pre-compiled script plugins to the rescue!

`buildSrc/src/main/kotlin/kotlin-conventions.kts`

```
plugins {
    kotlin("jvm")
}
dependencies {
    implementation(kotlin("stdlib-jdk8"))
}
tasks.withType<KotlinCompile> {
    kotlinOptions {
        freeCompilerArgs = listOf("-Xjsr305=strict")
        jvmTarget = "11"
    }
}
```

`subproject-a/build.gradle.kts`

```
plugins {
  `kotlin-conventions`
}
```

Tipps for authoring Plugins

Writing a plugin in Kotlin

Let's write a plugin that collects some names and then greets all those people.

`buildSrc/src/main/kotlin/GreetingPlugin.kt`

```kotlin
data class Who(project: Project) {
  val who = project.property<String>
}

val allWhos = mutableListOf<Who>()

fun greet(configure: Who.() -> Unit) {
  val who = Who(project)
  who.configure()
  allWhos.add(who.who.get())
}

project.tasks.add(GreetingTask::class.java, "greetings")
```

Using the plugin

`build.gradle.kts`

```
plugins {
  GreetingPlugin
}

greet {
  who.set("Hello World")
}
```

`build.gradle`

```
plugins {
  id 'GreetingPlugin'
}

// Doesn't work, type Who.() -> Unit is not accessible
greet {
  who = "Hello World"
}
```

General Advice

- Strongly type your plugin API even in Groovy
- Your plugin might be used from either Groovy or Kotlin (or even Java!)
- Don’t use language specific types
  - Avoid groovy.lang.Closure
  - Avoid Kotlin lambdas
- Use the built-in `org.gradle.api.Action` type

Fixed plugin

`buildSrc/src/main/kotlin/GreetingPlugin.kt`

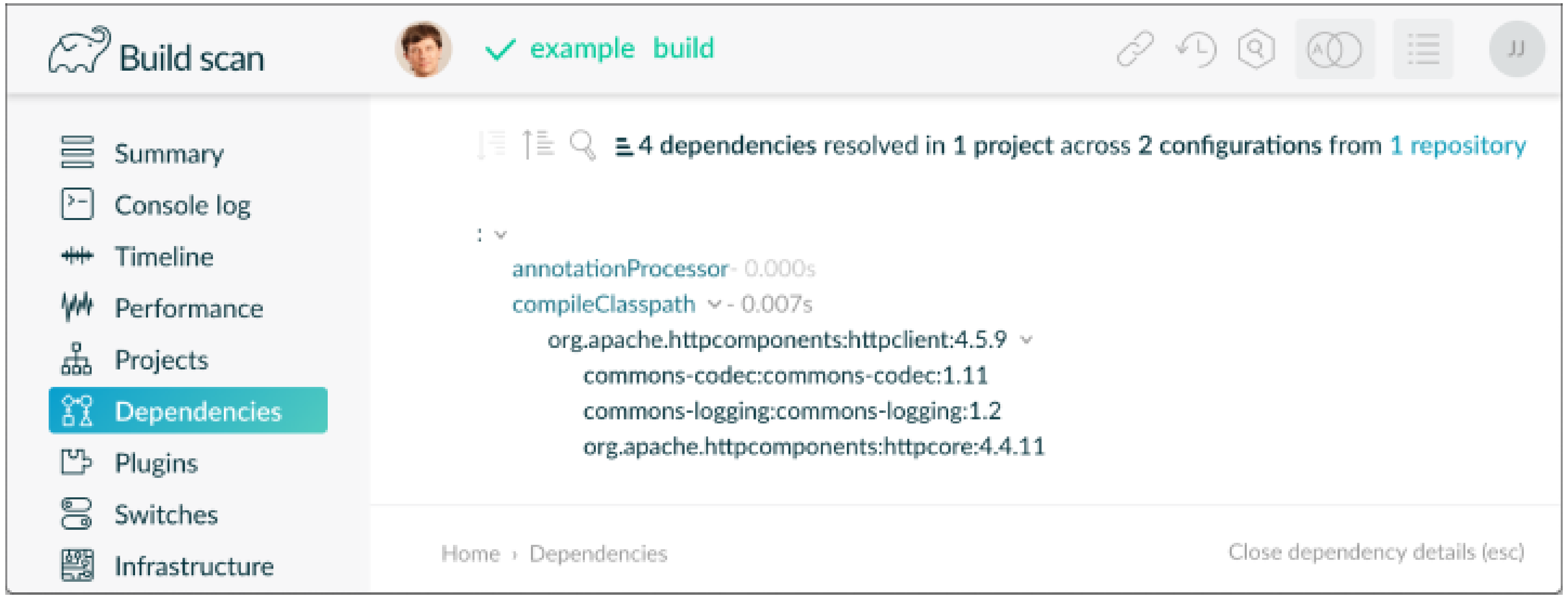
```
data class Who(project: Project) {
  val who = project.property<String>
}

val allWhos = mutableListOf<Who>()

fun greet(action: Action<Who>) {
  val who = Who(project)
  action.execute(who)
  allWhos.add(who.who.get())
}

project.tasks.add(GreetingTask::class.java, "greetings")
```

Maybe try a Build Scan with Gradle Enterprise



Build scan

example build

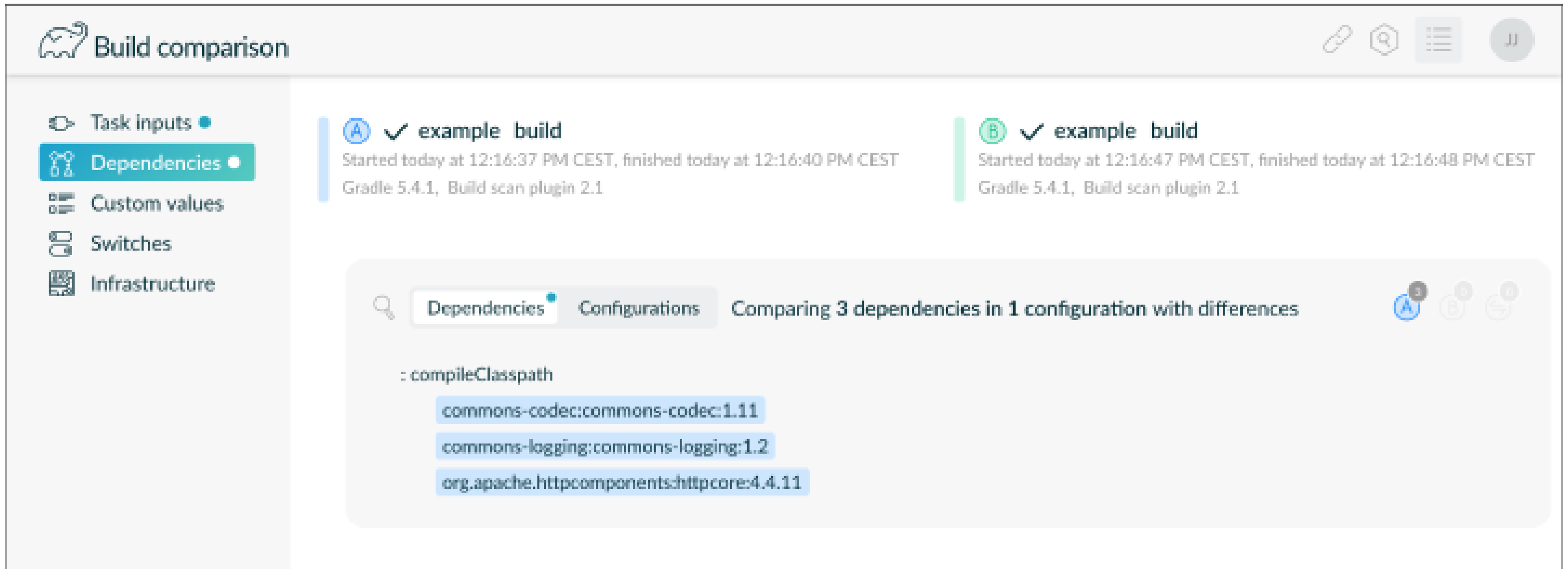
Summary
Console log
Timeline
Performance
Projects
Dependencies
Plugins
Switches
Infrastructure

4 dependencies resolved in 1 project across 2 configurations from 1 repository

```
:
  annotationProcessor - 0.000s
  compileClasspath - 0.007s
    org.apache.httpcomponents:httpclient:4.5.9
      commons-codec:commons-codec:1.11
      commons-logging:commons-logging:1.2
      org.apache.httpcomponents:httpcore:4.4.11
```

Home › Dependencies

Close dependency details (esc)



Build comparison

Task inputs
Dependencies
Custom values
Switches
Infrastructure

A example build
Started today at 12:16:37 PM CEST, finished today at 12:16:40 PM CEST
Gradle 5.4.1, Build scan plugin 2.1

B example build
Started today at 12:16:47 PM CEST, finished today at 12:16:48 PM CEST
Gradle 5.4.1, Build scan plugin 2.1

Dependencies Configurations Comparing 3 dependencies in 1 configuration with differences

```
: compileClasspath
    commons-codec:commons-codec:1.11
    commons-logging:commons-logging:1.2
    org.apache.httpcomponents:httpcore:4.4.11
```

Remember…

- With Gradle you model your project through a Java/Kotlin API
- The Kotlin DSL provides a staticly types facade to the underlying dynamic model
- Pre-compiled script script plugin are a conventient way to structure your build
- As a plugin author always use strong types but don’t expose language specific ones

Thank you

⍰⍰ britter.github.io/kotlin-everywhere-2019-cologne

🐘 github.com/gradle/gradle

⍰⍰ @BenediktRitter

🐱 @britter

🎙 autoweird.fm