Kotlin

Gradle: Build Automation Tool

@kotlin

Build Automation Tools

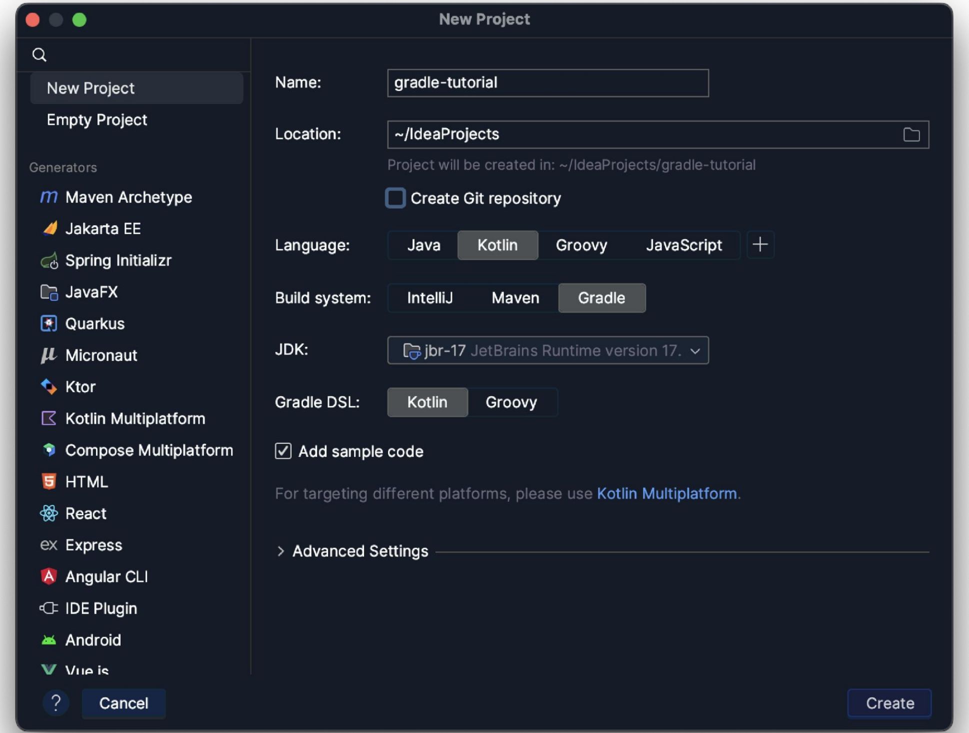
Build system – Software that automates the process of getting some kind of an artifact (executable, library) from the source code. Build systems can be used for:

- Configuring your build once and using it forever ~~copy-paste into new projects~~
- Unifying builds and reusing logic in various projects
- Dependencies management*
- Testing and verification
- Incremental builds*

Gradle

- is a build automation tool introduced in 2008;
- was targeted at Java (JVM overall) at first;
- inherited the idea of “Convention over Configuration” from Maven: Source of the project are in src folder, if not stated otherwise.
- inherited the Maven packages from Maven;
- used Groovy DSL at first, but now transition to Kotlin DSL is taking place;

New Gradle Project in IntelliJ



New Project

New Project
Empty Project

Generators
Maven Archetype
Jakarta EE
Spring Initializr
JavaFX
Quarkus
Micronaut
Ktor
Kotlin Multiplatform
Compose Multiplatform
HTML
React
Express
Angular CLI
IDE Plugin
Android
Vue.js

Name: gradle-tutorial

Location: ~/IdeaProjects

Project will be created in: ~/IdeaProjects/gradle-tutorial

Create Git repository

Language: Java [✓ Kotlin] Groovy JavaScript +

Build system: IntelliJ Maven [✓ Gradle]

JDK: jbr-17 JetBrains Runtime version 17.

Gradle DSL: [✓ Kotlin] Groovy

✓ Add sample code

For targeting different platforms, please use Kotlin Multiplatform.

Advanced Settings

Java bytecode

```
plugins {
    kotlin("jvm") version "1.8.0"
    application
}

group = "org.constructor.jetbrains.kotlin"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain(8)
}

application {
    mainClass.set("MainKt")
}
```

```
Failed to calculate the value

of task ':compileJava' property

'javaCompiler'. Unable to download

toolchain matching these requirements:

languageVersion=8, vendor=any,

implementation=vendor-specific

Unable to download toolchain. This

might indicate that the combination

(version, architecture, release/early

access, …) for the requested

JDK is not available. Could not read

'https:^/api.adoptopenjdk.net/v3/binary/latest/8/ga/mac/aarch64/jdk/h'

as it does not exist.
```

Java bytecode

I use JetBrains Runtime 17.0.5, hence my JDK version is 17, so to fix the problem I do:

```
kotlin {
    jvmToolchain(17)
}
```

But usually instead of jvmToolchain you will see:

```
tasks.withType<KotlinCompile> {
    kotlinOptions.jvmTarget = "1.8"
}
```

Build script

Gradle tries hard to look like a declarative build configuration, while using a procedural language (Groovy or Kotlin).

The first thing we see in build.gradle.kts is plugins { ^^. } block, which is a method of

```
class org.gradle.kotlin.dsl.KotlinBuildScript
:> abstract class ProjectDelegate
:> interface Project : Comparable, ExtensionAware, PluginAware
```

Project

This interface is the main API you use to interact with Gradle from your build file. From a Project, you have programmatic access to all of Gradle’s features. There is a one-to-one relationship between a Project and a “`build.gradle`” file

Lifecycle:

1. Assemble a Project object for each project
2. Create a `org.gradle.api.initialization.Settings` instance for the build
3. Evaluate the ”`settings.gradle`” script, if present, against the Settings object
4. Use the Settings object to create the hierarchy of Project instances.
5. Evaluate each Project by executing its ”`build.gradle`” file

Project: Settings

`Project → New → Module.`

A new folder (which is a new module) appears. It has its own `build.gradle.kts`, but no `settings.gradle.kts`, while a new line is added to the settings file in the root:

```
rootProject.name = "gradle-tutorial"
include("module")
```

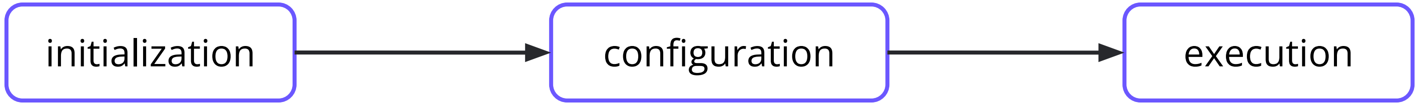
Sidenote: For a single-project build, a settings file is optional.

Project: Settings

A small addition to settings.gradle.kts:

```
rootProject.name = "gradle-tutorial"
include("module")
println("Initialization phase.")
```

```
Initialization phase.
> Task :prepareKotlinBuildScriptModel
UP-TO-DATE

BUILD SUCCESSFUL in 3s
```



Project: Tasks

A project is essentially a collection of Task objects. Each task performs some basic piece of work, such as compiling classes, or running unit tests, or zipping up a JAR file.

There are default tasks, tasks from plugins and your custom tasks. Custom tasks can be defined right in the build configuration.

Tasks can depend on each other. Gradle builds an acyclic directed graph of tasks.

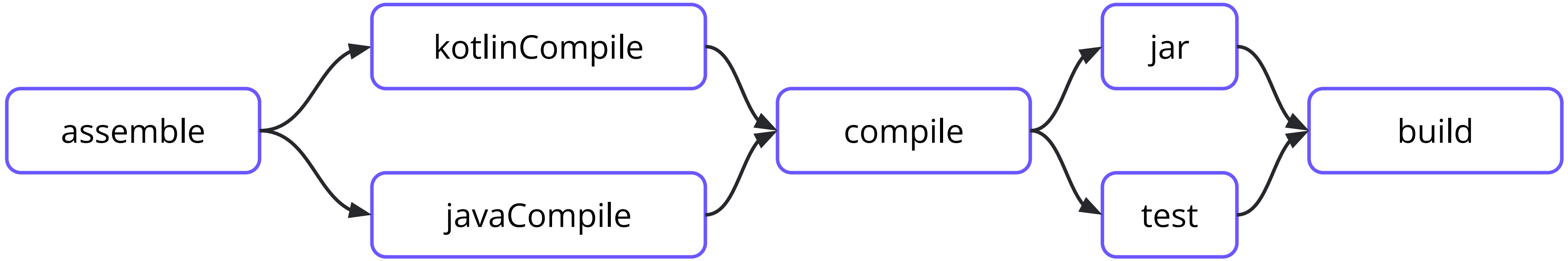
Tasks have inputs and outputs. Gradle cashes results of tasks. If on a new run inputs hash has not changed, Gradle uses the previous result there: UP-TO-DATE. If hash changed, but it is present in cache, then: FROM-CACHE. Otherwise, it still tries to reuse previous results.

Project: Tasks

```kotlin
tasks.register("targetTask") {
    group = "useless"
    dependsOn(tasks.named("dependencyTask"))
    println("${this.name}, configuration")
    doFirst {
        println("${this.name}, first in execution")
    }
}

tasks.register("dependencyTask") {
    println("${this.name}, configuration")
    doFirst {
        println("${this.name}, first in execution")
    }
    doLast {
        println("${this.name}, last in execution")
    }
}
```

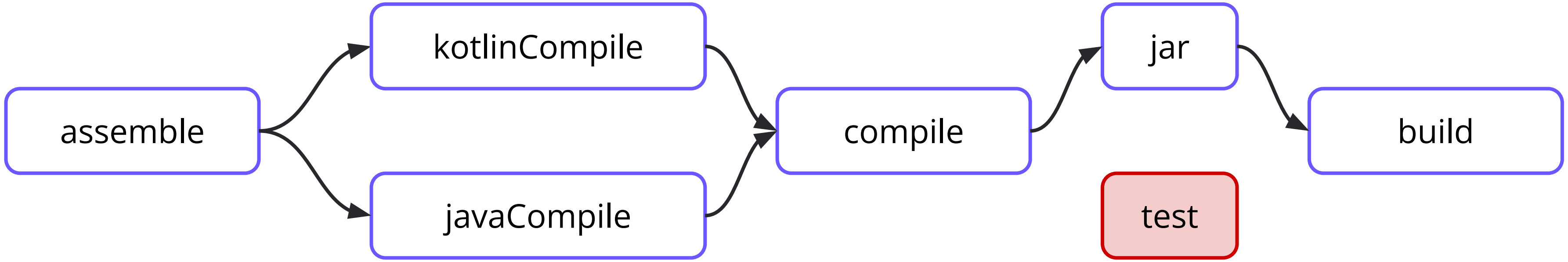
```
./gradlew :targetTask

Executing 'targetTask'...

> Configure project :
targetTask, configuration
dependencyTask, configuration

> Task :dependencyTask
dependencyTask, first in execution
dependencyTask, last in execution

> Task :targetTask
targetTask, first in execution
```

Tasks



```
./gradlew build
```

Tasks



```
./gradlew build -x test
```

Build without running tests.

Project: Tasks

Configuration language is Kotlin (or Groovy) DSL, so everything is possible.

You can add behaviour to already existing tasks.

```kotlin
fun Task.printName() = println("Hi! My name is ${this.name}")

val task3 = tasks.register("thirdTask") {
    doFirst { printName() }
}

tasks.filter { task -> task.group?.let { it != "useless" } ?: false }
      .forEach {
          it.dependsOn(task3)
      }

tasks.test {
    useJUnitPlatform()
}
```

Plugins

Most of the useful features are added by plugins.

Plugins add new tasks (e.g. JavaCompile), domain objects (e.g. SourceSet), conventions (e.g. Java source is located at src/main/java) as well as extending core objects and objects from other plugins.

There are binary plugins and script plugins. Binary plugins are usually an external plugin jar. Script plugins are typically used within a build.

First, Gradle resolves the plugin, and then it needs to apply the plugin to the target, usually a Project `(Plugin.apply(target: T))`.

Plugins

Gradle provides the core plugins (e.g. ApplicationPlugin, application) as part of its distribution.

Applying a community plugin:

```
plugins {
    kotlin("jvm") version "1.8.0" // id("org.jetbrains.kotlin.jvm") version "1.8.0"
    id("io.gitlab.arturbosch.detekt") version "1.21.0" apply false // later apply it in a subproject
}
```

There are also plugins that are build from `buildSrc` and then applied by id as though they are community plugins.

Custom plugin repositories can be added via pluginManagement `{ … }` block in `settings.gradle.kts`.

Plugins

```
// our build.gradle.kts
plugins {
    kotlin("jvm") version "1.8.0"
    application
}

kotlin { // provided by kotlin("jvm") community plugin
    jvmToolchain(8)
}

application { // provided by application core plugin
    mainClass.set("MainKt")
}
```

We applied two plugins to our project.

Plugins

```
// our build.gradle.kts
plugins {
    kotlin("jvm") apply false
    kotlin("plugin.serialization") apply false
}
allProjects {
    apply {
        plugin("kotlin")
    }
}
```

We applied only the Kotlin plugin to all modules in the project eagerly.

Plugins

```
// our build.gradle.kts
plugins {
    kotlin("jvm") apply false
    kotlin("plugin.serialization") apply false
}

…

val ignored = listOf("common")

configure(subprojects.filter { it.name !in ignored }) {
    apply {
        plugin("kotlinx-serialization")
    }
}
```

We applied the `kotlinx-serialization` plugin to all modules except `common`.

Dependency Management

Gradle will locate the declared dependencies in repositories, which are declared as local directories or a remote repositories. This process is called dependency resolution.

Once resolved, the resolution mechanism stores the underlying files of a dependency in a local cache (the local Maven repository).

Gradle handles dependency resolution in case there are conflicting transitive dependencies. This behaviour is also customizable.

Dependency Management: Repositories

```
repositories {
    mavenCentral()
    maven {
        url = uri("https://your.company.com/maven")
            credentials {
                username = USER_NAME
                password = PASSWORD
            }
        }
    flatDir {
        dirs("libraries")
    }
}
```

Dependency Management: Dependencies

```
val ktor_version: String by project

dependencies {
    // the string notation, e.g. group:name:version
    implementation("io.ktor:ktor-server-core-jvm:$ktor_version")
    // map notation + api (internals are accessible)
    api("io.ktor", "ktor-server-netty-jvm", ktor_version)
    // dependency on another project
    implementation(project(":neighborProject"))
    // putting all jars from 'libs' onto compile classpath
    implementation(fileTree("libs"))
    // test dependency
    testImplementation(kotlin("test")) }
```

Dependency Management: Dependencies

```
// settings.gradle.kts
val utilitiesRepo = "https:^/github.com/JetBrains-Research/plugin-utilities.git"
val utilitiesProjectName = "org.jetbrains.research.pluginUtilities"

sourceControl {
    gitRepository(URI.create(utilitiesRepo)) {
        producesModule("$utilitiesProjectName:plugin-utilities-core")
    }
}

// build.gradle.kts
val utilitiesProjectName = "org.jetbrains.research.pluginUtilities"

dependencies {
    implementation("$utilitiesProjectName:plugin-utilities-core") {
        version {
            branch = main
        }
    }
}
```

We have 1 implementation dependency from the `main` branch of a GitHub repository.

Properties

Properties are used to to configure behavior of Gradle itself and specific projects. In order of highest to lowest precedence:

1. Command-line flags such as `--build-cache`;
2. `gradle.properties` in `GRADLE_USER_HOME` directory;
3. `gradle.properties` in the project's directory up to the root project directory;
4. `gradle.properties` in Gradle installation directory (`~/.gradle/gradle.properties`);
5. Environment variables;

To see all available properties: `./gradlew properties`

Properties

`projectDir/module/build.gradle.kts`

```kotlin
tasks.register("printProperty") {
    val prop: String? by project
    doLast {
        println(prop ?: "Not set")
    }
}
```

```
./gradlew :module:printProperty → Not set
```

If we add `prop="Prop set"` to `projectDir/module/gradle.properties`, then:

```
./gradlew :module:printProperty → Prop set
./gradlew -Pprop="Override prop":module:printProperty → Override prop
```

Wrapper

The recommended way to execute any Gradle build is with the help of the Gradle Wrapper.

The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary.

You can build, run, etc projects with the Wrapper locally without global Gradle on the laptop.

> One way to upgrade the Gradle version is manually change the `distributionUrl` property in the Wrapper’s `gradle-wrapper.properties` file.
>
> – *Gradle Documentation*

Version catalog

You can configure all plugins, dependencies, and versions in one place. Inside `settings.gradle.kts` via `dependencyResolutionManagement { versionCatalogs { … } }` or in a special `TOML` file, conventionally named `libs.versions.toml` and stored in the gradle folder.

`TOML` file has a special structure:

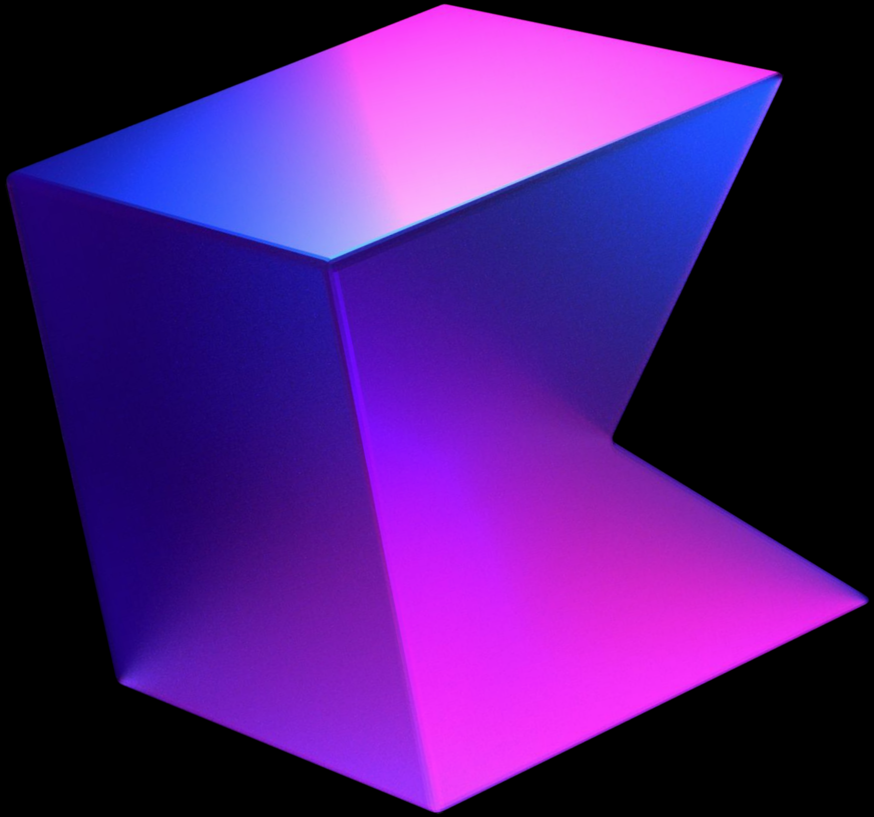
```
[versions]
kotlin = "1.7.10"

[libraries]
junit-jupiter-api = {
     module = "org.junit.jupiter:junit-jupiter-api", version.ref = "junit-jupiter"
}

[plugins]
kotlin-jvm = { id = "org.jetbrains.kotlin.jvm", version.ref = "kotlin" }
```

Version catalog

```
// build.gradle.kts
plugins {
    val kotlinVersion = libs.versions.kotlin.get()
    id(libs.plugins.kotlin.jvm.get().pluginId) version kotlinVersion
        apply false
}

allProjects {
    dependencies {
        testImplementation(rootProject.libs.junit.jupiter.api)
    }
}
```

We included 1 plugin and 1 dependency via the `TOML` file

Thanks!



@kotlin