Gradle

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

What is a build system?

A build system is a system that manages the process of delivering software. This includes compilation, linking, testing, and packaging.

There are many options for build configuration tools [1]. Examples of popular build systems include:

- Ant/Maven for Java
- Cargo for Rust
- Cmake/Scons/Bazel for C++.

Characteristics of a useful build system:

- It provides consistency in builds and build results.
- It is expressive so that you can define any custom tasks e.g., zip a file.
- You can automate the build process to avoid user errors.
- It integrates with other systems so that you can delegate responsibility e.g., remote test under a different OS.

What is Gradle?

Gradle is a modern build system for Java/Kotlin.

- It's a cross-platform and programming language agnostic build system.
- It's open source and has a large community of users.
- It's the default when working with Android, and popular in the Kotlin and Java ecosystems.

Three main areas of functionality:

- Managing build tasks: Manage build tasks e.g., compile and link, run tests.
- Build configuration: Define and manage how these tasks are executed.
- Dependency management: Manage external libraries and dependencies.

Getting Started with Gradle

Creating a Gradle project

To setup your code to build with Gradle, you need a Gradle “project” - a folder structure with configuration files describing how to build your source code.

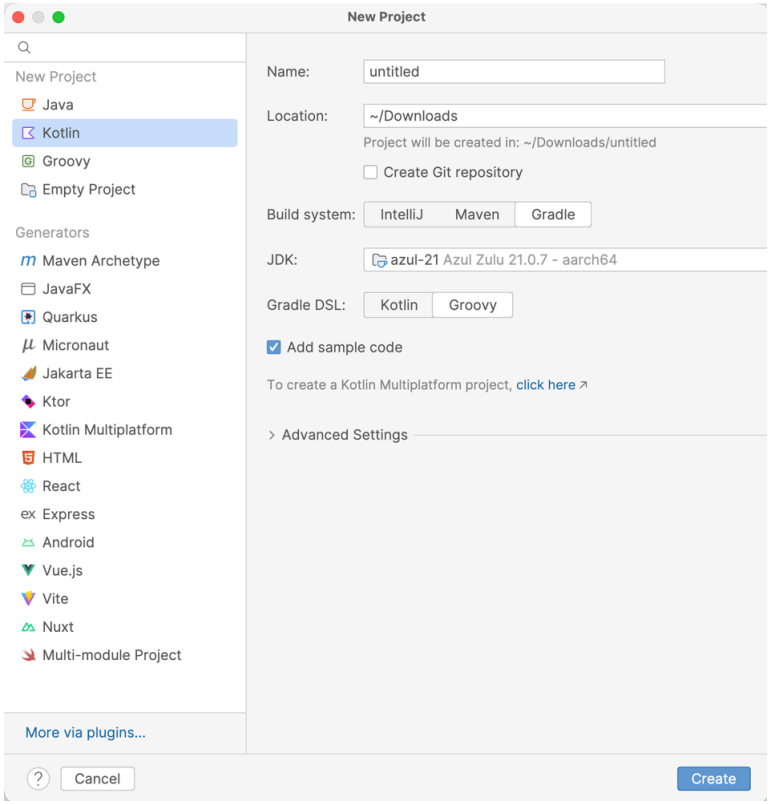


Figure 1: Gradle is bundled with IntelliJ IDEA, so you can use the New Project wizard to create a starting project.

Project Structure

Your project structure will vary based on the type of project (more on this later). Here's the structure of a basic, single target Gradle project.

```
.
├── build.gradle.kts
├── empty.iml
├── gradle
│   └── wrapper
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── src
    ├── main
    │   ├── kotlin
    │   │   └── Main.kt
    │   └── resources
    └── test
        ├── kotlin
        └── resources
```

The contents of a Gradle build directory include:

- `build.gradle.kts` is the main config file.
- `empty.iml` is the IntelliJ config file.
- `gradle/` contains gradle wrapper config.
- `gradlew` & `gradlew.bat` are scripts to execute Gradle commands.
- `settings.gradle.kts` is a top-level project config file.
- src/ contains source code and unit tests.

Using Gradle

Gradle Tasks

`Tasks` are Gradle build commands that you can run. They can be executed from the command-line using the Gradle scripts, or directly from IntelliJ.

Command gradle tasks, run from the command-line:

```
$ ./gradlew clean
$ ./gradlew build
$ ./gradlew run
```

There are many other tasks that are supported. For a complete list, run:

```
$ gradle tasks
```

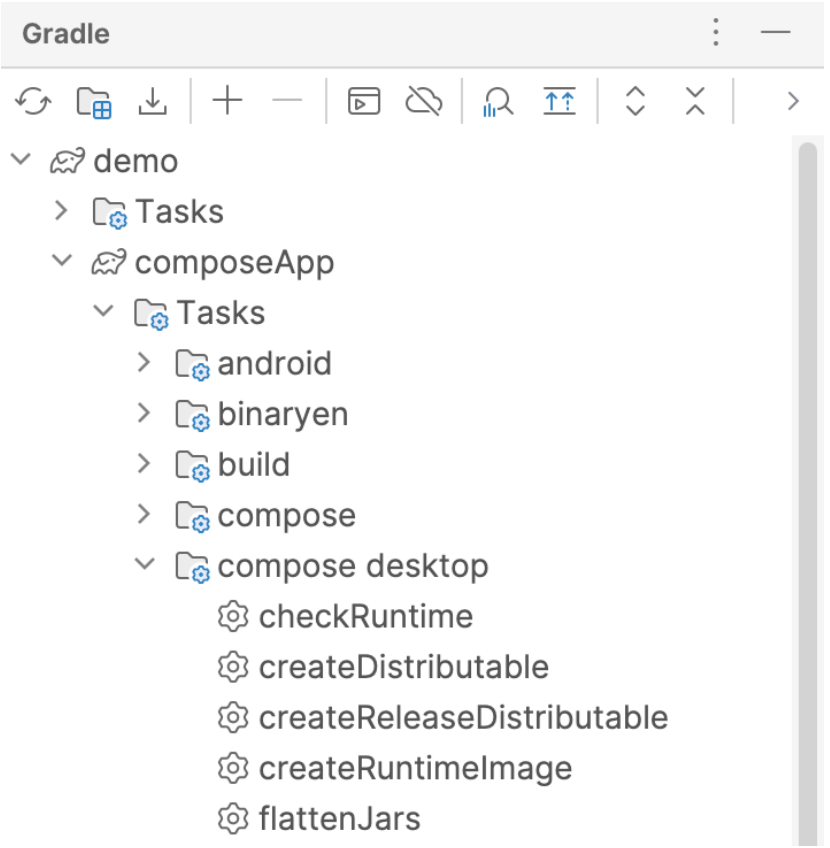


Figure 2: View-Gradle will bring up a list of Gradle tasks, grouped by target.

Gradle Wrappers

At the top-level of your project's directory structure are two scripts that we saw earlier:

- `gradlew` for macOS and Linux users, and
- `gradlew.bat` for Windows users

These are Gradle wrapper scripts that are generated when you create your project. You can use them to **run Gradle tasks without having to install Gradle on your machine**.

When run, the scripts will download Gradle for you, install it, and then run the commands using that version of Gradle.

```
$ ./gradlew build
```

> **Note**
>
> Is this a good idea? Why not just install Gradle manually?

Build Configuration

```
.
├── build.gradle.kts
├── empty.iml
├── gradle
│   └── wrapper
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── src
    ├── main
    │   ├── kotlin
    │   │   └── Main.kt
    │   └── resources
    └── test
        ├── kotlin
        └── resources
```

Let's discuss project configuration.
The two main configuration files are:

build.gradle.kts

- this is the main configuration file that contains project settings.
- this is at the top-level of the project and applied to the entire project.
- it is possible to have multiple modules (e.g., app/, service/), each with its own `build.gradle.kts` file specific to that type of module.

settings.gradle.kts - project level.

- contains global settings that apply to all modules.

Build Configuration

build.gradle.kts

```
plugins {
    kotlin("jvm") version "2.3.10"
}

group = "ca.uwaterloo"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies { ... }

kotlin {
    jvmToolchain(25)
}
```

This is a standard desktop build configuration.

Modify it to:

- Add a new dependency (i.e. library)
- Add a new plugin (i.e. custom Gradle tasks)
- Update the version number of a product release.

Don’t expect to create the perfect config file right-away.

- Start with the one generated by IntelliJ IDEA.
- Modify as you add dependencies or make changes.

Build Configuration

settings.gradle.kts

```
plugins {
  id("org.gradle.toolchains.foojay-resolver-convention") version "1.0.0"
}

rootProject.name = "My Project Name"
```

This is the top-level configuration file. By default:

- It includes a plugin that can download and install a JVM, if needed, to compile and test this build.
- It defines the project name of your project, which is useful when exporting modules.

Build Configuration

Dependencies

Dependencies are external libraries to provide functionality e.g., networking, user interfaces.

- They need to be downloaded and added to your project to be useful.
- In a simple project, you would just download them and `#include` them in your source code.

In a larger projects with many dependencies, we need to be concerned about versioning and compatibility.

- Making sure that you have the correct version of a library,
- Including dependencies that a library might need (`transitive dependencies`).
- Making sure that the library is compatible with the rest of your software, and that it doesn't introduce any security vulnerabilities.

In Gradle, you specify your dependencies in your build scripts, and Gradle will download and manage them for you.

Adding Libraries

You can search Maven Central or use a package manager like this klibs.io.

Pay attention to supported platforms! KMP projects need versions of libraries that will work on all supported platforms. We don't want to introduce platform-specific code if we can avoid it.

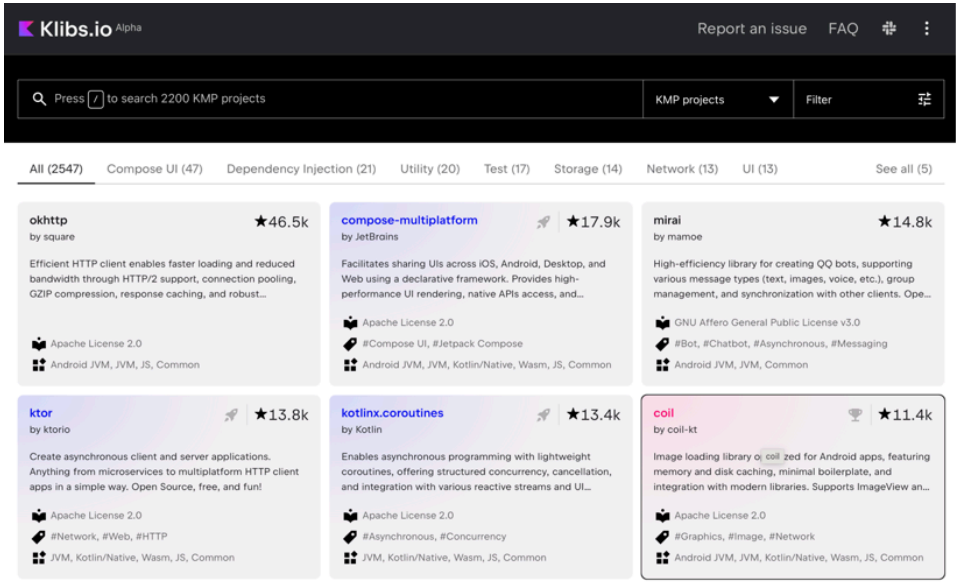


Figure 3: klibs.io is recommended because it only includes KMP or multi-platform librariess, so you can instantly tell if your OS/platform is supported!

Adding Libraries

Adding Dependencies

You add a specific module or dependency by adding it into the dependencies section of the `build.gradle.kts` file. Dependencies need to be specified using this syntax:

```
group-name: module-name: version-number
```

We can often copy and paste the dependency line from the package information page directly into our `build.gradle.kts`. For example, the klibs.io coil entry includes instructions on how to add it to your Gradle project.

```
dependencies {
    implementation("io.coil-kt.coil3:coil-compose:3.1.0")
}
```

Adding Libraries

Version Catalogs

One challenge to using a lot of dependencies is keeping track of the versions of libraries that you are using. Gradle has a feature called a **version catalog**, which is a centralized file that contains a list of libraries and their versions.

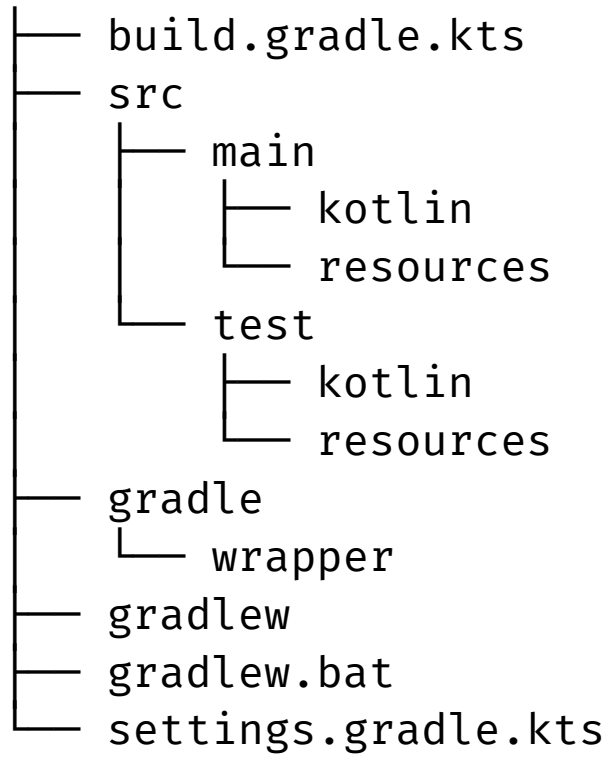
- Gradle will automatically keep versions up-to-date using this file.
- The version catalog is contained in a file `libs.versions.toml` in your gradle/ project directory.
- You use the dependencies defined in the version catalog in your build config files.

See the Gradle documentation for more information.

Types of Projects

Single-Target Build

There is a single build target e.g., JVM desktop. This is our example from earlier.



The top-level module is defined in the root of the project.

- Configuration files are at the top-level.
- Source tree is also at the root.

This is a single module.

Types of Projects

Multi-Project Build

```
├── app/
│   ├── build.gradle.kts
│   └── src
│       ├── main
│       └── test
├── gradle/
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── util/
    ├── build.gradle.kts
    └── src
        ├── main
        └── test
```

Our project has two modules:

- `app`
- `util`

There is a `build.gradle.kts` is specific to each module. This means each one can have a different target, and different dependencies.

- e.g., `app` might produce a desktop application, while `util` might be a module that we want to reuse in a different project.

There are other options to target multiple platforms that we'll see

- e.g., KMP.

Types of Projects

Kotlin Multiplatform (KMP) Build

```
├── build.gradle.kts
├── composeApp
│   ├── build.gradle.kts
│   └── src
│       ├── androidMain
│       ├── commonMain
│       ├── desktopMain
│       └── iosMain
├── gradle
├── iosApp
│   ├── Configuration
│   │   └── Config.xcconfig
│   ├── iosApp
│   │   ├── Assets.xcassets
│   │   ├── ...
│   └── iosApp.xcodeproj
│   └── project.pbxproj
├── local.properties
└── settings.gradle.kts
```

This project is meant for multi-platform situations where you want to share code using `Compose`.

In a KMP project, your source code into split across two top-level projects.

- `composeApp` includes common code. It is further subdivided by target platform: android, common, desktop and iOS.
- `iosApp` includes the iOS project and configuration files, used to build and package using Xcode and other macOS tools.

Bibliography

[1] Wikipedia, “List of Build Tools.” [Online]. Available: https://en.wikipedia.org/wiki/List_of_build_automation_software