Build Systems & Gradle

CS 346 Application Development

Build Systems

A **build system** is a system that manages the process of delivering software. This includes compilation, linking, testing, packaging and any other required steps. Build systems have become more complex and capable over time.

- e.g., 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.

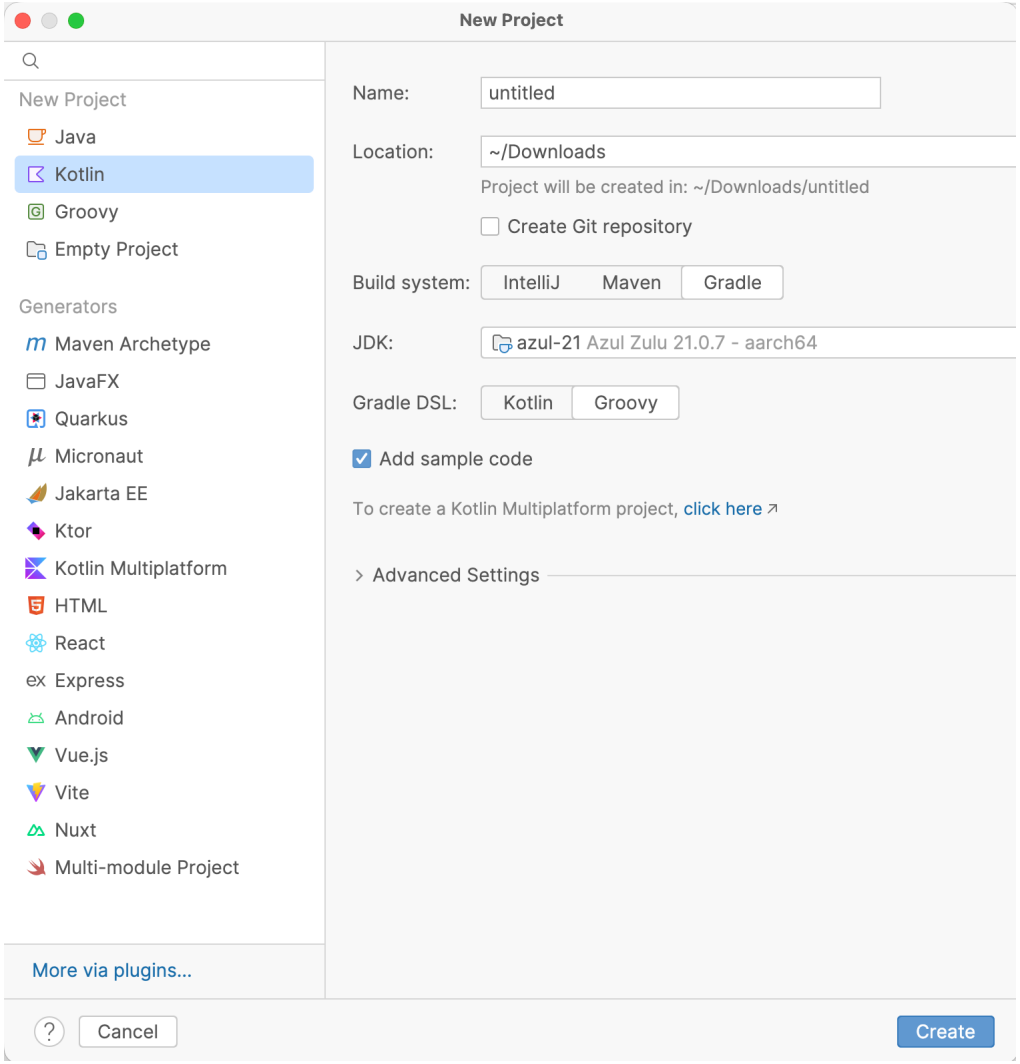
- It's popular in the Kotlin and Java ecosystems, used for Android projects.
- It's cross-platform and programming language agnostic.
- It's open source and has a large community of users.

Three main areas of functionality:

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

Getting Started

Gradle project structure



Gradle is bundled with IntelliJ IDEA.

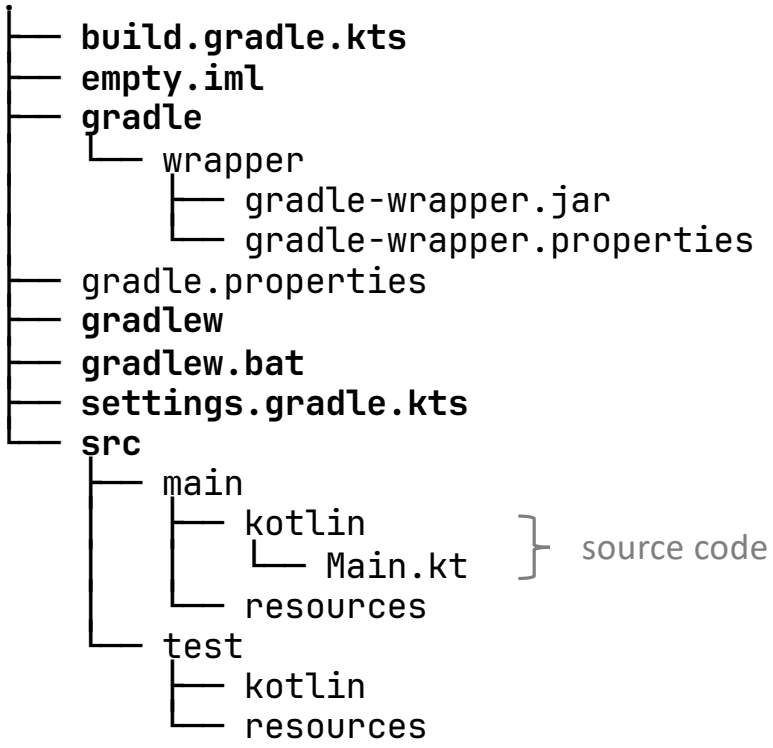
To create a Gradle project:

1. Use the `Project Wizard`.
2. Select `Gradle` for your Build system.
3. Select `Kotlin` for your Gradle DSL.
4. Click `Create`.

See the course website:

Reference > Programming
> Create a Gradle Project

Basic Project Structure



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.

settings.gradle.kts is a top-level project config file.

src: contains source code

- `src/main/kotlin` code module
- `src/test/kotlin` unit test module

Build Tasks

How to execute Gradle tasks.

Gradle Tasks

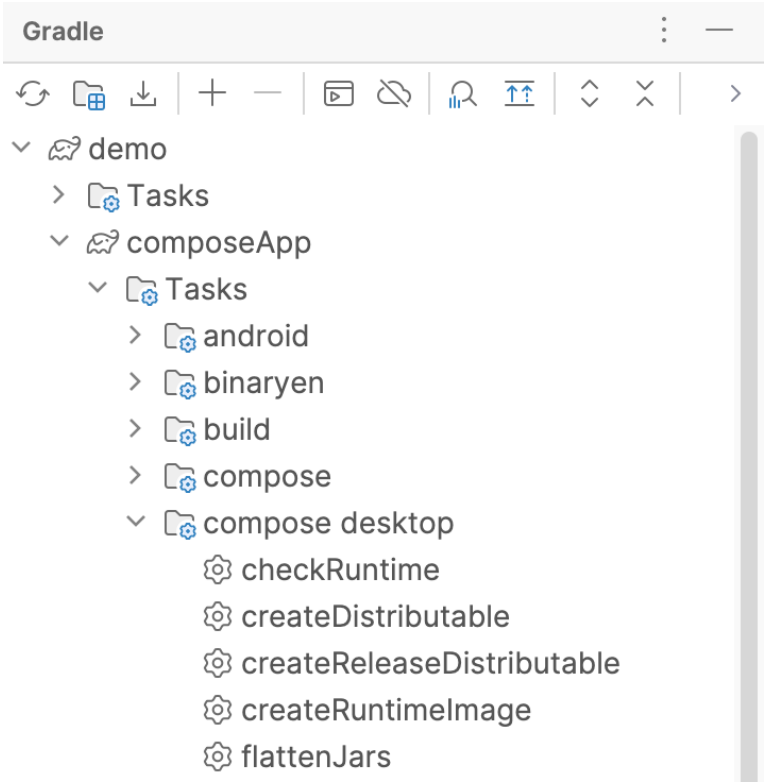
Tasks are built-in commands that you can run, that are specific to your project.

They can be executed either:

- From the command-line
- From the Gradle menu in Intellij IDEA

Command gradle tasks, run from the command-line:

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



In IntelliJ IDEA: View > Tool Windows > Gradle

Gradle Wrapper

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

- `gradlew` for Unix users, and
- `gradlew.bat` for Windows users

These are *Gradle wrapper scripts.* You can use them to run Gradle tasks without having to install Gradle on your machine.

- Pass them command-line arguments.
- The scripts will download Gradle for you, install it, and then run the commands using that version of Gradle.

```
$ ./gradlew build
```

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

Plugins

- Gradle comes with a small number of predefined tasks. You can add additional tasks to your project as `plugins`.
- A plugin is a collection of related tasks that have been bundled
  - `java` plugin - adds language support, and
  - `application` plugin - adds support for running a console app.

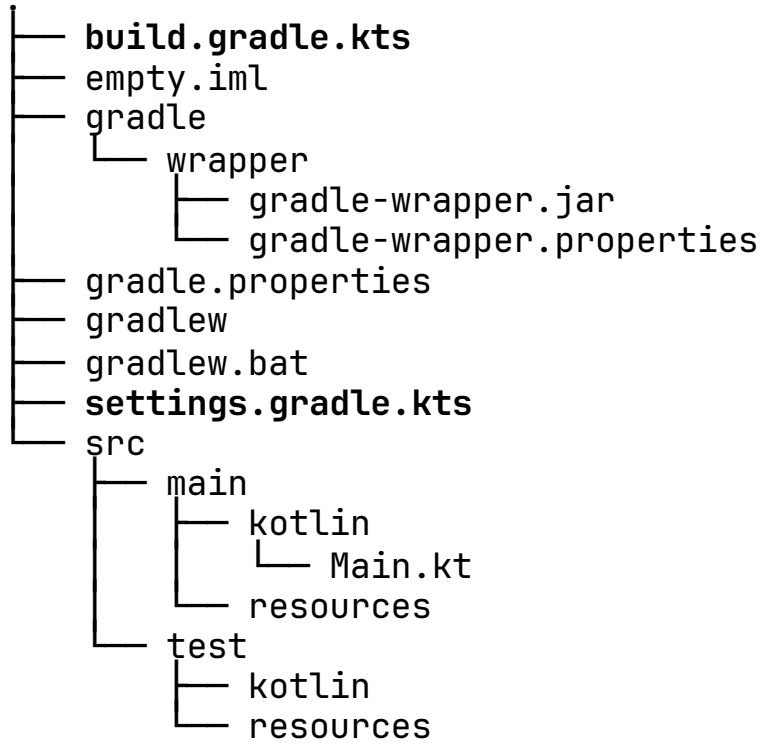
You add other plugins in your `build.gradle.kts` file.

```
plugins {
  application
   kotlin("jvm") version "2.0.10"
}
```

Build Configuration

How to manage your build configuration.

Config files



build.gradle.kts - module specific

- It is possible to have multiple modules (e.g., app/, service/). Each of these would have its own `build.gradle.kts` file specific to that type of module.
- This example has a single module, at the root.

settings.gradle.kts - project level.

- It contains settings that apply to all modules.

settings.gradle.kts

This is the top-level configuration file. You don't need to modify this for single-target projects.

```
// list any plugins that you want to use across all modules
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.5.0"
}

// top-level descriptive name
rootProject.name = "project-name"
```

settings.gradle.kts

build.gradle.kts

This is the detailed build configuration. Modify it to:

- Add a new dependency (i.e. library)
- Add a new plugin (i.e. custom 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.

```
// needed for desktop
plugins {
    kotlin("jvm") version ”2.0.10"
}

// product release info
group = "org.example"
version = "1.0.0"

// location to find libraries
repositories {
    mavenCentral()
}

// add libraries here
dependencies {
    testImplementation(`org.jetbrains.kotlin:kotlin-test`)
}

tasks.test {
    useJUnitPlatform()
}

// java version
kotlin {
    jvmToolchain(21)
}
```

build.gradle.kts

Dependencies

How to manage project dependencies.

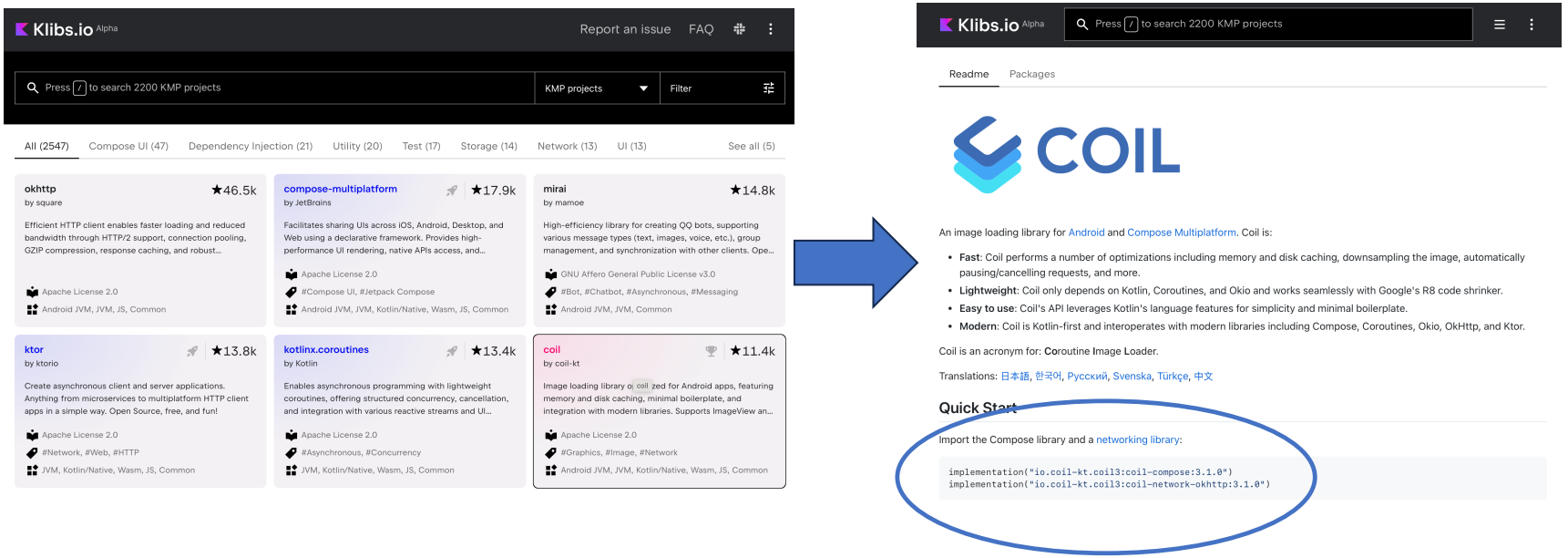
What are dependencies?

In this context, 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.
- A large challenge of any build system is managing these dependencies. i.e.,
  - Making sure that you have the correct version of a library,
  - Including dependencies *that* library might need (called *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.
  - Gradle will download them from an online repository as part of your build process.

Finding dependencies

- You can search Maven Central or use a package manager like this klibs.io.
- Pay attention to supported platforms: does it work on your platform?



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`

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

group-name module-name version

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 `version catalogs`, 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.
  - In Gradle 7.x or later, 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.

https://docs.gradle.org/current/userguide/version_catalogs.html

gradle/libs.versions.toml

```
[versions]
guava = "32.1.3-jre"
junit-jupiter = "5.10.1"

[libraries]
guava = { module = "com.google.guava:guava", version.ref = "guava" }
junit-jupiter = { module = "org.junit.jupiter:junit-jupiter",
version.ref = "junit-jupiter" }
```

build.gradle.kts

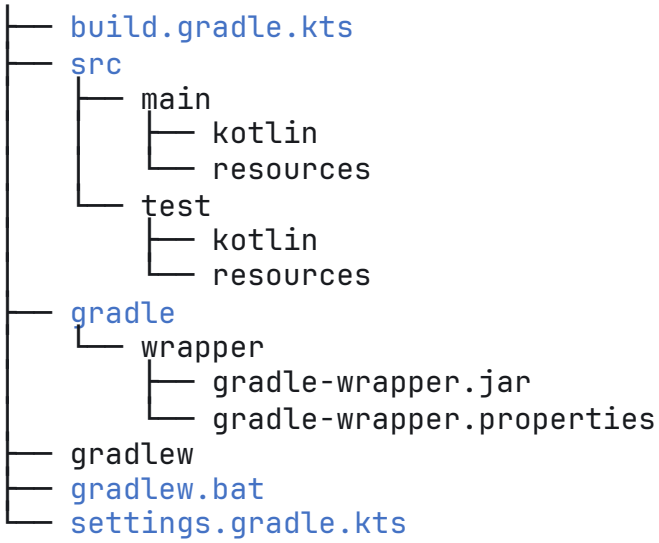
```
dependencies {
    // This dependency is used by the application.
    implementation(libs.guava)
}
```

Types of Gradle projects

Getting started with a new project.

Single Project Structure

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, loosely defined.

IntelliJ: New Project > Kotlin

Build system: Gradle
Gradle DSL: Kotlin
Generate multi-module build: **unchecked**

Multi-Project Structure

A “better” structure moves the source code into multiple modules.

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

`app` is the first module.
`util` is a second module.

- build.gradle.kts is specific to each module.

Why 2 modules?

- Different platforms e.g., android, jvm
- Different purposes e.g., one could be published as a library, the other as an app.

IntelliJ: New Project > Kotlin

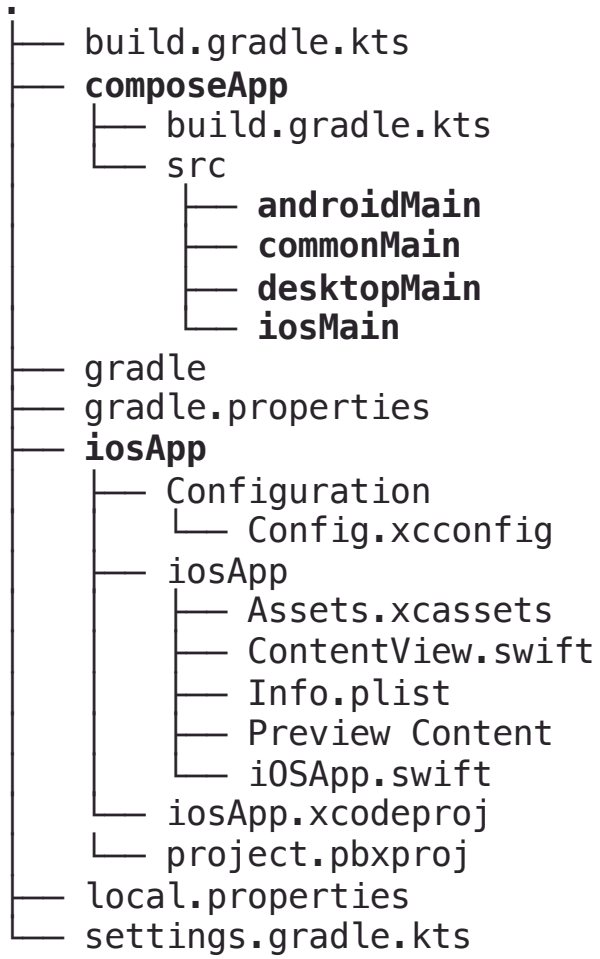
Build system: Gradle
Gradle DSL: Kotlin
Generate multi-module build: **checked**

KMP Project Structure

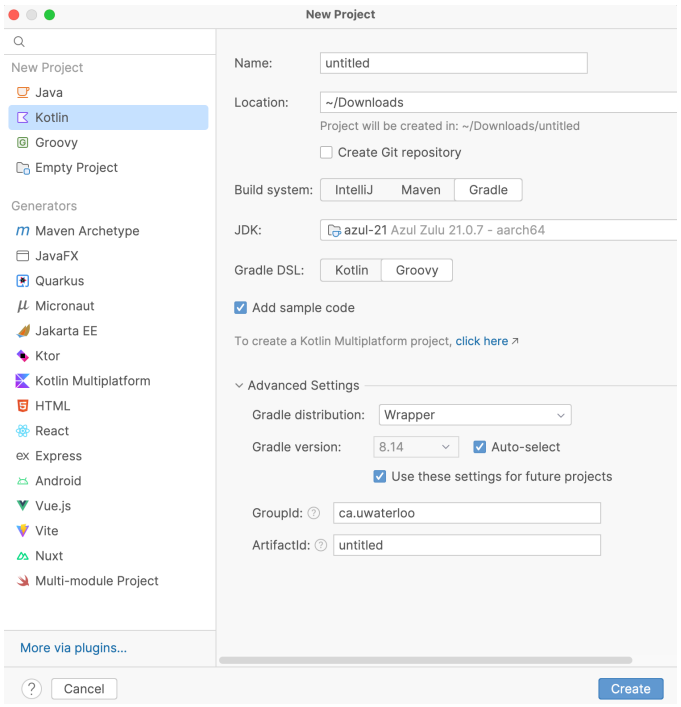
The project breaks down the source code into two main projects.

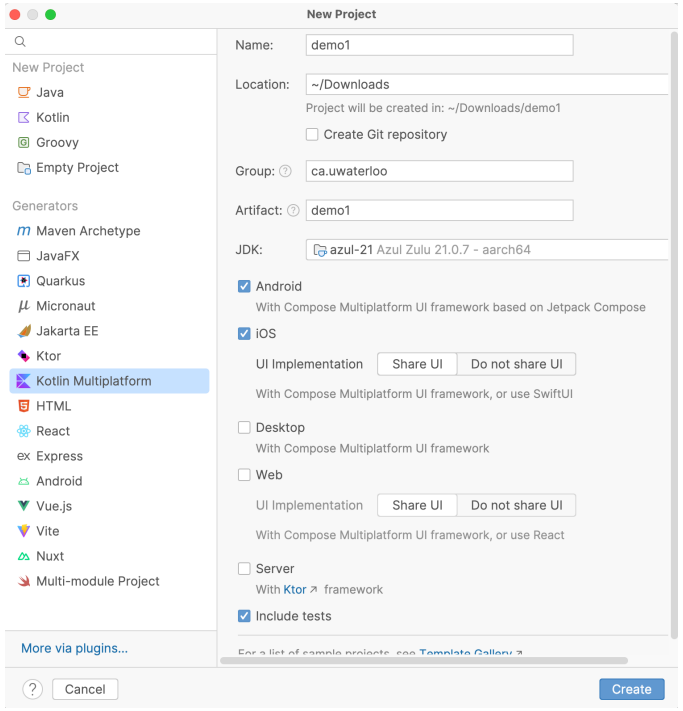
- **composeApp** includes all Compose code. It is further split into android, common, desktop and iOS.
  - This is where you add source code.
- **iosApp** includes the iOS project and configuration files, used to build and package using Xcode and other macOS tools.
  - Integration point for Kotlin/iOS.
  - You probably shouldn't touch this!

IntelliJ: New Project > Kotlin Multiplatform



Instructions: How to create a Gradle project





Reference > Programming > Create a Gradle Project

Reference

- Gradle.org. 2024. Gradle User Manual.
- Gradle.org. 2025. Version Catalogs.
- Philipp Lackner. 2025. The Ultimate Gradle Kotlin Beginner's Crash Course
- Tom Gregory. 2024. Gradle Build Bible.