

Build Systems & Gradle

CS 346 Application
Development

Deploying Applications

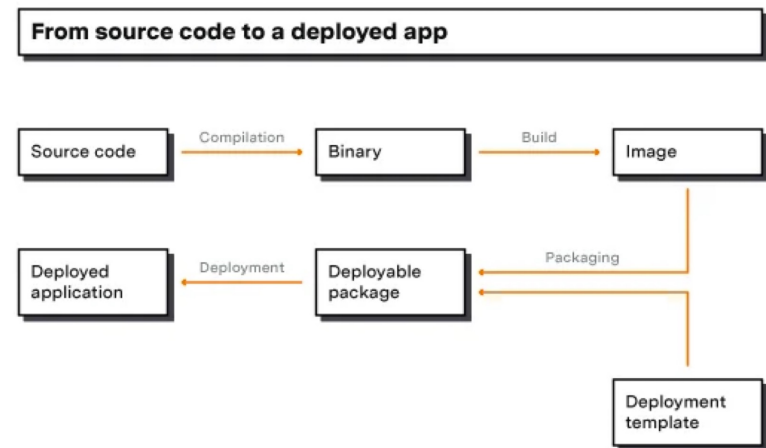
As a developer, your final goal isn't just working code; it's an application that can be installed and executed by your users.

To achieve this, you will need to perform several steps:

- Check library versions, resources.
- Compile your application.
- Run tests, ensure everything works.
- Build installers for your users.

You don't want to do this manually!

- These steps are all very complex.
- Any of them can introduce errors.



Build Systems

A **build system** is a system that manages the tasks required to build software, including compilation, linking, automated testing, packaging.

- e.g., Maven for Java; Cargo for Rust; Cmake/Scons/Bazel for C++.

Characteristics of a *useful* build system:

- It provides **consistency** in builds so that you get consistent results.
- It is **expressive** so that you can define any custom tasks
 - e.g., generate a ZIP file, or convert some documentation to include in the installation.
- You can **automate** many of the steps to avoid user errors.
- It **integrates with other systems** so that you can delegate responsibility
 - e.g., remote test under a different OS.

Why we don't use `GNU Make`

`Make` is quick-and-easy for simple projects, but it's not the best choice for large, complex projects.

- **It required build dependencies to be explicitly defined.**
 - Libraries must exist on the build machine and be defined in your `makefile`.
- **Make is tied to the underlying environment** of the build machine.
 - It's difficult to isolate make's runtime behaviour from the underlying environment.
 - e.g., \$LIB using the user's environment variable to track library locations.
- **Performance is poor.** Make doesn't scale well to large projects.
- **The language itself isn't very expressive** and cannot easily be extended.
- **It's very difficult to fully automate** and integrate with other systems.

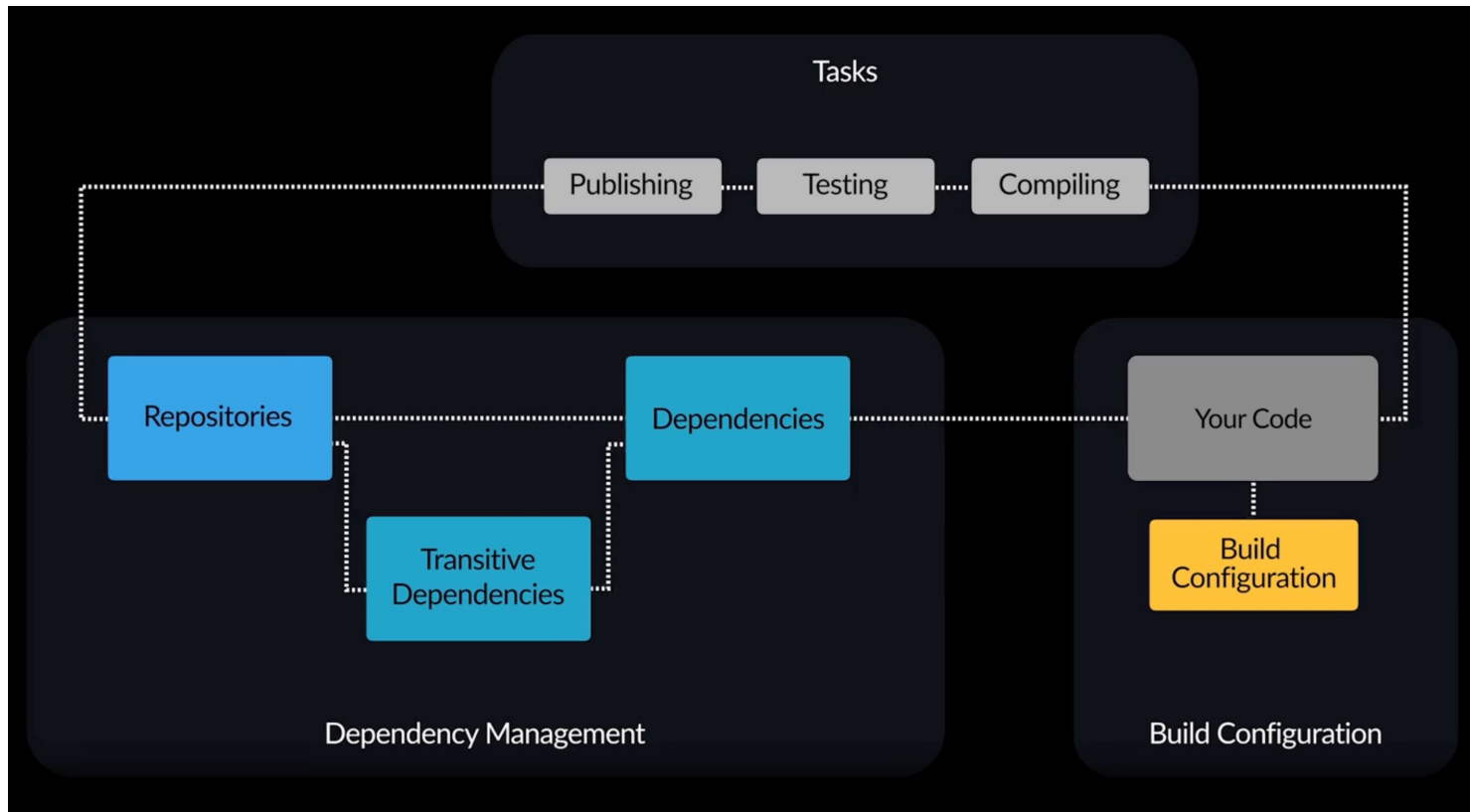
Why Gradle?

Gradle is a modern build system for Java/Kotlin.

- It's popular in the Kotlin and Java ecosystems.
- It's the official build tool 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:** Built-in support for discrete tasks that you will need to perform. e.g., downloading libraries; compiling code; running unit tests and so on.
2. **Build configuration:** Define and manage how these tasks are executed.
3. **Dependency management:** Manage external libraries and dependencies.



The pillars of a build system: managing code and dependencies, tasks that define actions to take, and configuration scripts that determine how to run these tasks.

Getting Started

Gradle project structure

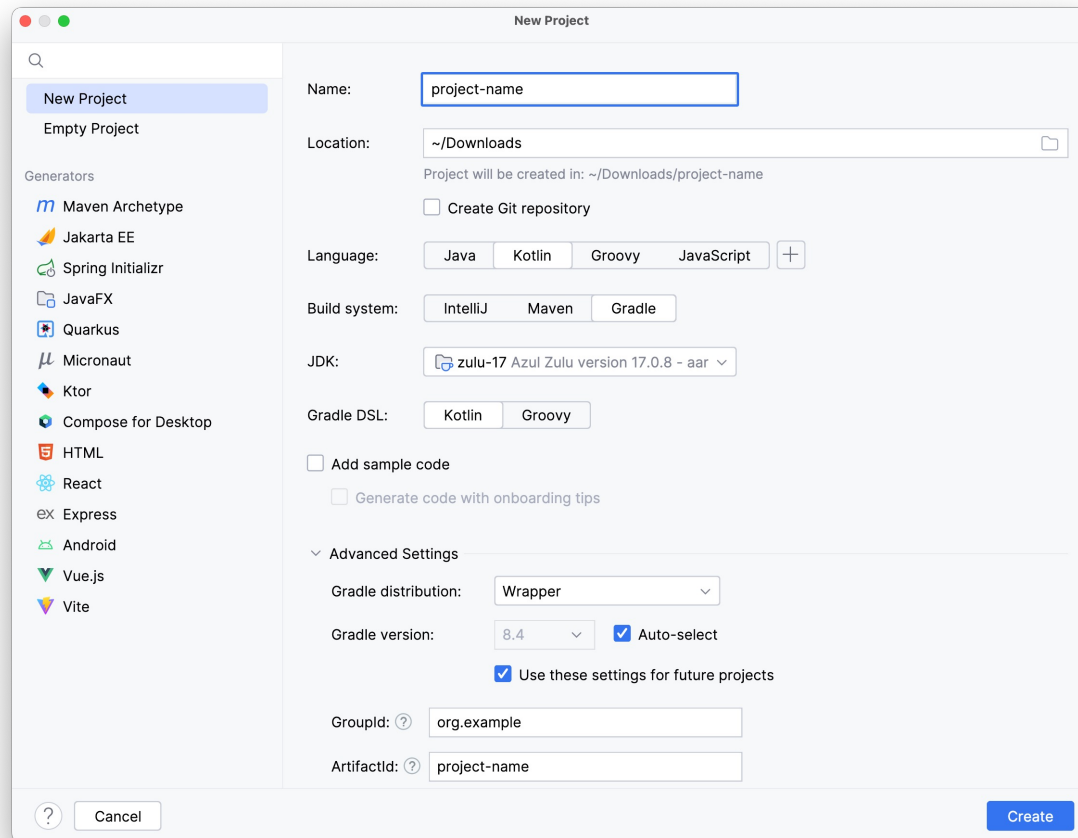
Creating a project.

Gradle is a command-line application that you use to create and interact with a project. A project consists of:

- A directory structure with a specific layout, and
- Configuration files describing the build configuration and any custom tasks.

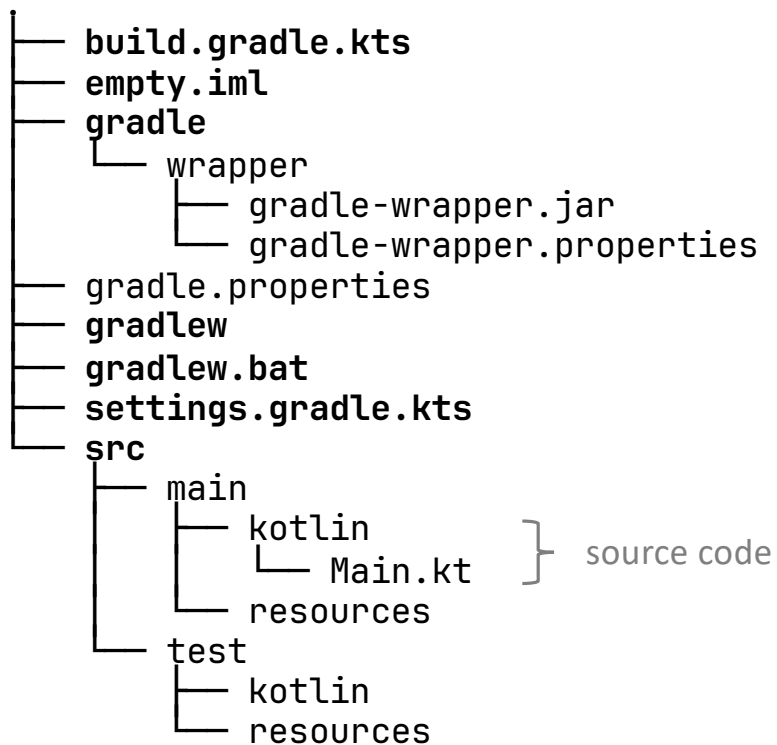
You can create this project in IntelliJ IDEA or Android Studio, or by using the `gradle init` task.

- We *highly recommend* using the appropriate project wizard in your IDE – see the course website.



Creating a project in IntelliJ IDEA. Choose Kotlin as your Language, Gradle for your Build system, and Kotlin for your Gradle DSL.

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.

What are build tasks?

Projects often have complex build requirements that include a series of steps that need to be performed. For example, you might need to:

1. Import dependencies e.g., libraries,
2. Compile your source code,
3. Run tests to make sure it works properly,
4. Build a distributable package.
5. Deploy to a server.



Any build system needs to support a wide range of steps like this, and it should allow you to define how they will be performed.

Gradle calls these actions ``build tasks``. Your application probably has many these that need to be run, in the correct order.

Running Tasks

- From the command-line, run `gradlew` (or `gradlew.bat` with a task name.
- e.g.,
 - \$./gradlew clean
 - \$./gradlew build
 - \$./gradlew run
- The tasks that are available are specific to the type of project you are working with.

```
$ ./gradlew tasks
```

```
> Task :tasks
```

```
-----  
Tasks runnable from root project 'gradle'  
-----
```

```
Application tasks
```

```
-----  
run - Runs this project as a JVM application
```

```
Build tasks
```

```
-----  
assemble - Assembles the outputs of this project.
```

```
build - Assembles and tests this project.
```

```
buildDependents - Assembles and tests this project and all dependent projects.
```

```
buildNeeded - Assembles and tests this project and all projects it depends on.
```

```
classes - Assembles main classes.
```

```
clean - Deletes the build directory.
```

```
jar - Assembles a jar archive containing the classes of the 'main' feature.
```

```
testClasses - Assembles test classes.
```

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?

Gradle Wrapper Config

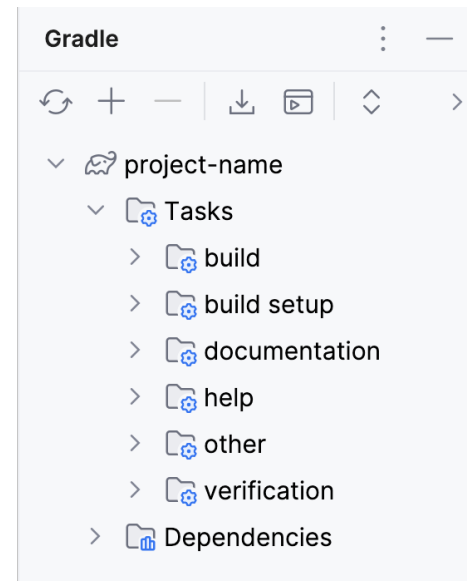
- The Gradle project configuration (`gradle/gradle-wrapper.properties`) lists the version of Gradle to be used for your project. It's a text file, with contents (something like) this:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0.2-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

- To specify the version of Gradle being used in your project, change the `distributionURL` line to the correct version e.g., Gradle 8.0.2.

IntelliJ Support

- IntelliJ IDEA supports Gradle.
- Gradle tool window shows all tasks grouped by type.
- Common tasks include:
 - gradlew help
 - gradlew tasks
 - gradlew clean
 - gradlew build
 - gradlew run



View > Tool Windows > Gradle will open the Gradle window, listing the supported tasks for your project.

Plugins

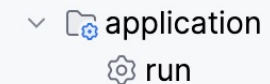
- Gradle comes with a small number of predefined tasks. You can add additional tasks to your project as plugins.
- A plugin is a collection of related tasks that have been bundled e.g., java plugin adds tasks for compiling Java code.
 - **Core plugins:** Included with Gradle by default, and they provide functionality for many projects. e.g.,
 - `java` plugin (adds language support) and
 - `application` plugin (adds support for running a console app).
 - **Community Plugins:** These are plugins that are created by the community and are not included by default.
 - See [Gradle Plugin Portal](#).

You specify plugins in your `build.gradle.kts` file (see next section).

build.gradle.kts

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

← Application plugin adds the run task.



```
application {  
    mainClass = "ca.uwaterloo.cs346.MainKt"  
}
```

← Configuration details for that plugin
i.e., package.name of the class that
contains the main method; what we
run when the run task executes.

```
group = "ca.uwaterloo.cs346"  
version = "1.0.0"
```

← Used for
packaging (later)

Build Configuration

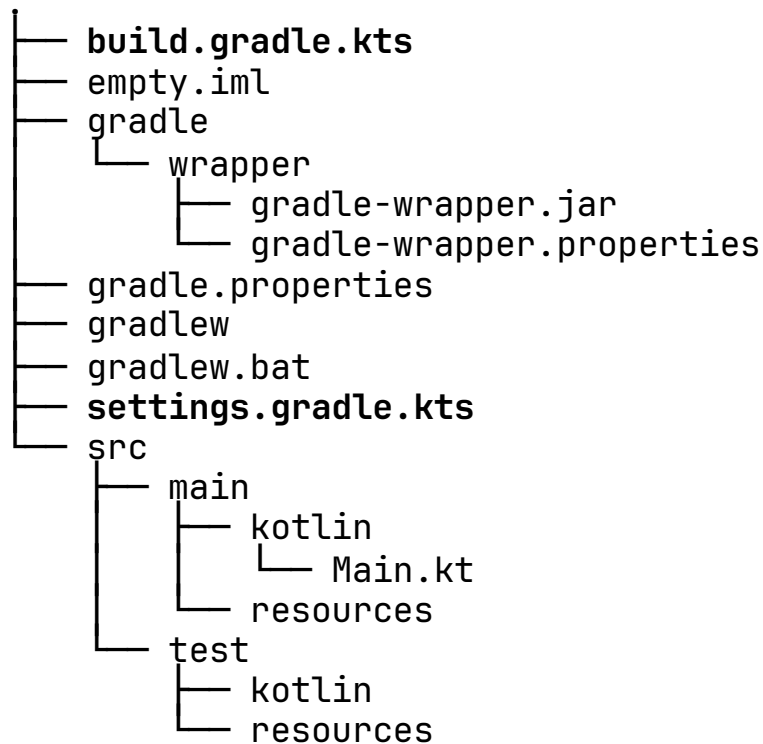
How to manage your build configuration.

What is build config?

Once you have tasks defined, you need some way to configure and control how they are executed.

- You could write custom scripts e.g., bash shell scripts to execute these tasks, but they are challenging to maintain.
- Gradle provides a way to define tasks in **build configuration files** and then run them with a single command.
- This makes it easy to build complex projects and ensures that the build process is consistent.
 - Gradle uses a Domain Specific Language (DSL) to define build scripts. We can use Kotlin DSL for our build scripts!

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. You might need to modify this file 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 (or gradle init)
 - 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?

When we write software, we often rely on external libraries to provide functionality that we don't want to write ourselves. e.g., networking, user interfaces. These libraries are dependencies of our application.

- 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.

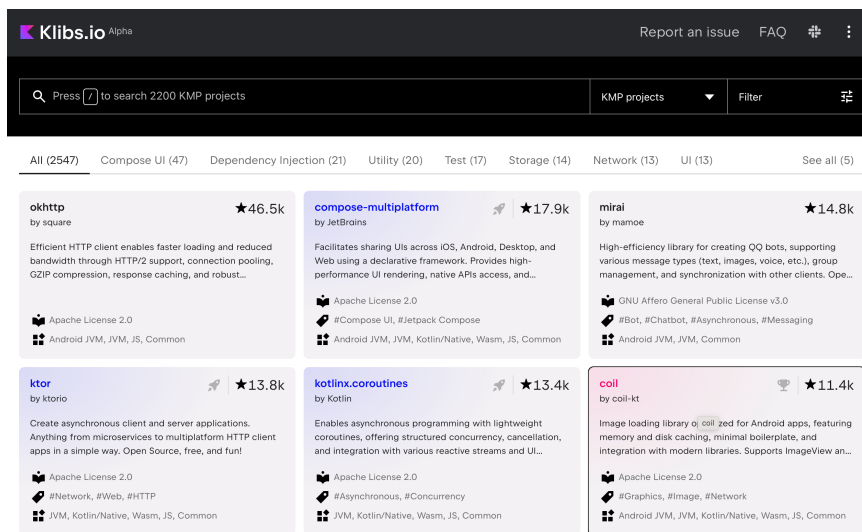
Where do we find these dependencies?

A **repository** is a location where libraries are stored and made available; these can be private (e.g. hosted in your company) or public (e.g. hosted and made available to everyone).

- Typically, a repository will offer a large collection of libraries across many years of releases, so that a package manager is able to request a specific version of a library and all its dependencies.
- The most popular Java/Kotlin repository is [mavenCentral](#), and we'll use it with Gradle to import any external dependencies that we might require.

Finding dependencies

- You can search Maven Central or use a package manager like this klibs.io.
- Each package will include details of how to import and use it.



Klibs.io Alpha

Report an issue FAQ

Press [7] to search 2200 KMP projects

KMP projects Filter

All (2547) Compose UI (47) Dependency Injection (21) Utility (20) Test (17) Storage (14) Network (13) UI (13) See all (5)

okhttp by square ★46.5k
Efficient HTTP client enables faster loading and reduced bandwidth through HTTP/2 support, connection pooling, GZIP compression, response caching, and robust...

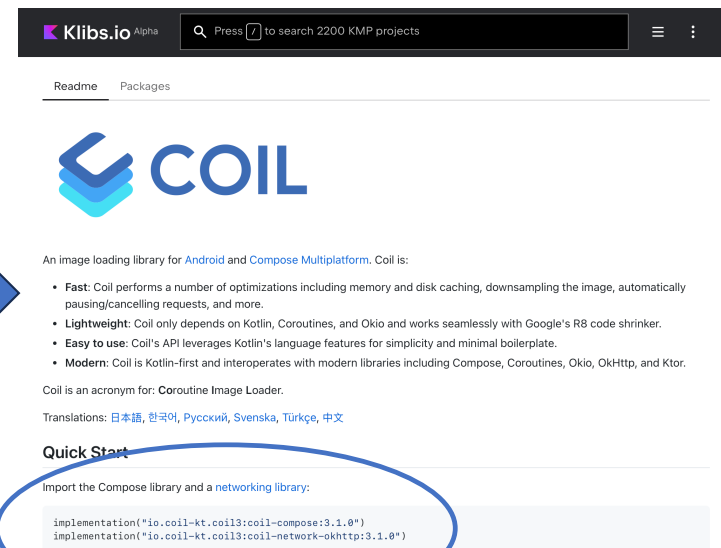
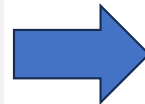
compose-multiplatform by JetBrains ★17.9k
Facilitates sharing UIs across iOS, Android, Desktop, and Web using a declarative framework. Provides high-performance UI rendering, native APIs access, and...

mirai by mamoe ★14.8k
High-efficiency library for creating QQ bots, supporting various message types (text, images, voice, etc.), group management, and synchronization with other clients. Ope...

ktor by ktorio ★13.8k
Create asynchronous client and server applications. Anything from microservices to multiplatform HTTP client apps in a simple way. Open Source, free, and fun!

kotlinx.coroutines by Kotlin ★13.4k
Enables asynchronous programming with lightweight coroutines, offering structured concurrency, cancellation, and integration with various reactive streams and UI...

coil by coil-kt ★11.4k
Image loading library for Kotlin Multiplatform (KMP) for Android apps, featuring memory and disk caching, minimal boilerplate, and integration with modern libraries. Supports ImageView an...



Klibs.io Alpha

Press [7] to search 2200 KMP projects

Readme Packages

COIL

An image loading library for **Android** and **Compose Multiplatform**. Coil is:

- **Fast**: Coil performs a number of optimizations including memory and disk caching, downsampling the image, automatically pausing/cancelling requests, and more.
- **Lightweight**: Coil only depends on Kotlin, Coroutines, and Okio and works seamlessly with Google's R8 code shrinker.
- **Easy to use**: Coil's API leverages Kotlin's language features for simplicity and minimal boilerplate.
- **Modern**: Coil is Kotlin-first and interoperates with modern libraries including Compose, Coroutines, Okio, OkHttp, and Ktor.

Coil is an acronym for: **C**oroutine **I**mage **L**oader.

Translations: [日本語](#), [한국어](#), [Русский](#), [Svenska](#), [Türkçe](#), [中文](#)

Quick Start

Import the Compose library and a **networking library**:

```
implementation("io.coil-kt:coil-compose:3.1.0")
implementation("io.coil-kt:coil3:coil-network-okhttp:3.1.0")
```

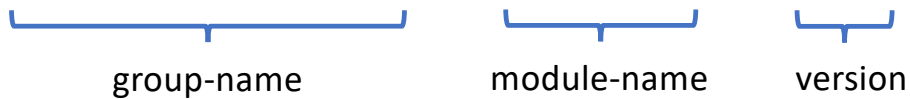
Adding Dependencies

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")  
}
```



The diagram illustrates the structure of the dependency string `io.coil-kt.coil3:coil-compose:3.1.0` within the `implementation()` call. Blue brackets are placed under the string to identify its parts: the first bracket under `io.coil-kt.coil3` is labeled `group-name`; the second bracket under `coil-compose` is labeled `module-name`; and the third bracket under `3.1.0` is labeled `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

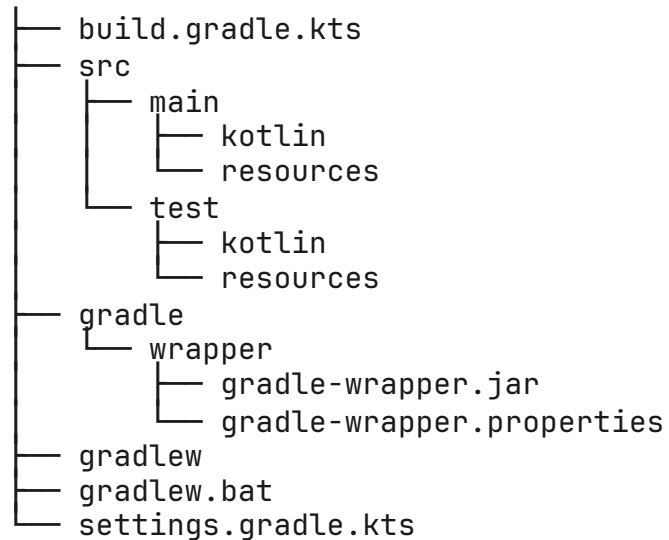
```
dependencies {
    // This dependency is used by the application.
    implementation(libs.guava)
}
```

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.

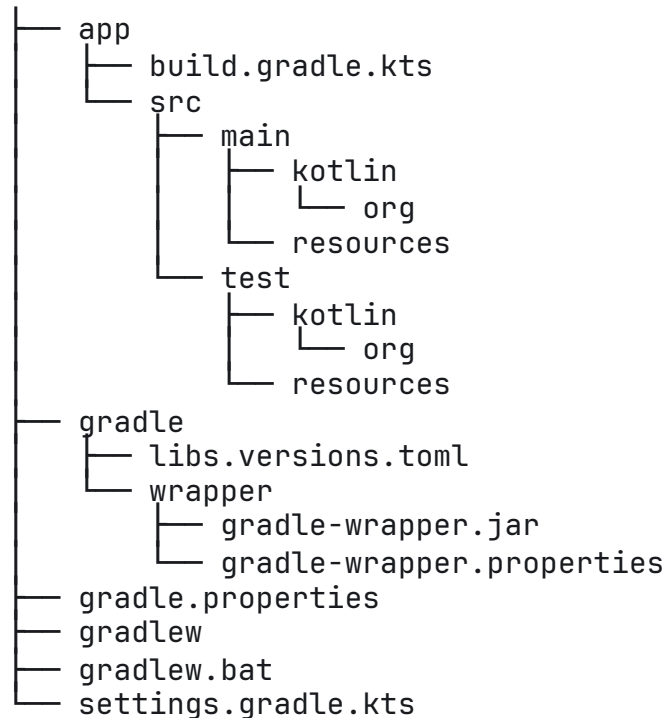


Configuration files are at the top-level.
Source tree is also at the root.

This is a single module, loosely defined.

Single Project w/ Module Structure

A “better” structure moves the source code into a single module.



`app` is the module name.

- build.gradle.kts is specific to the module.
- settings.gradle.kts remains root level

Makes it easy to create another module at the root level of the project! This can be useful later.

A second module could be used for

- A second build target.
- Code that you wish to split out into a library.
- Shared code between modules.

Multi-Project Structure

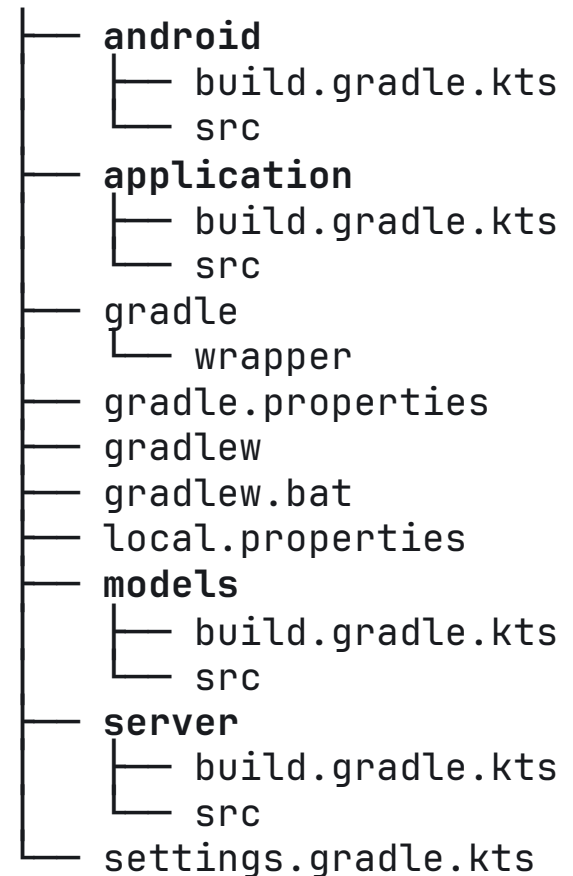
More than one module, each with its own `build.gradle.kts` file

- e.g. `android`, `application`, `models`, `server`

Shared `settings.gradle.kts` file to specify which modules to include.

Each module has its own configuration.

- **`android`**: config files to build native Android.
- **`application`**: config to build desktop/jvm.
- **`models`**: shared code, doesn't build a target.
- **`server`**: builds a Ktor server (JAR file).



KMP Project Structure

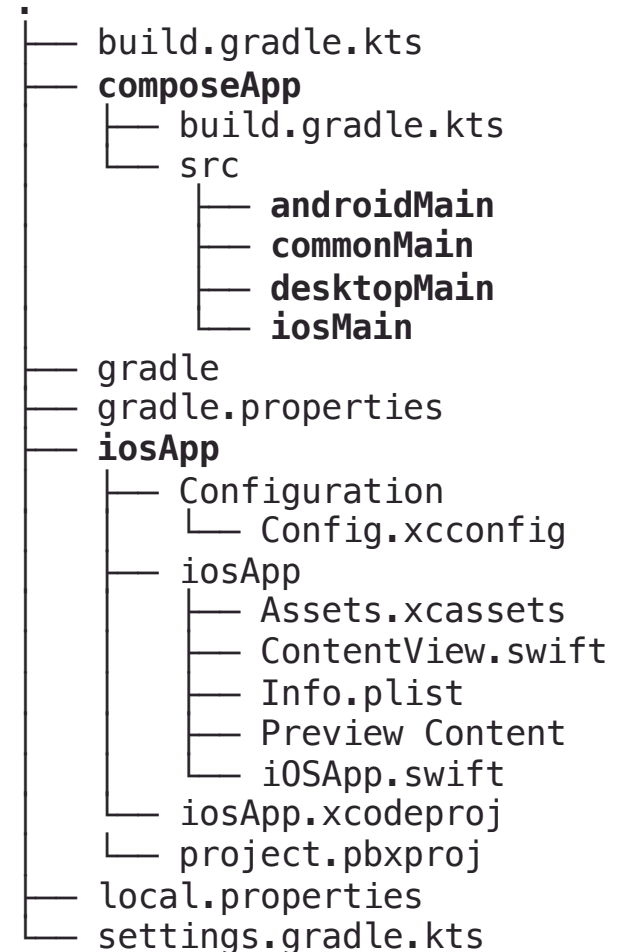
The project breaks down the source code into two main projects.

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!



How to create Gradle projects?

- 1. Command line
 - gradle init
 - Only recommended for very simple projects!
- 2. IntelliJ IDEA
 - Kotlin project (+/- multiplatform option), or
 - Kotlin Multiplatform.
- 3. Android Studio
 - Android project template.



See the Getting-Started section of the website for a walkthrough.

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](#).