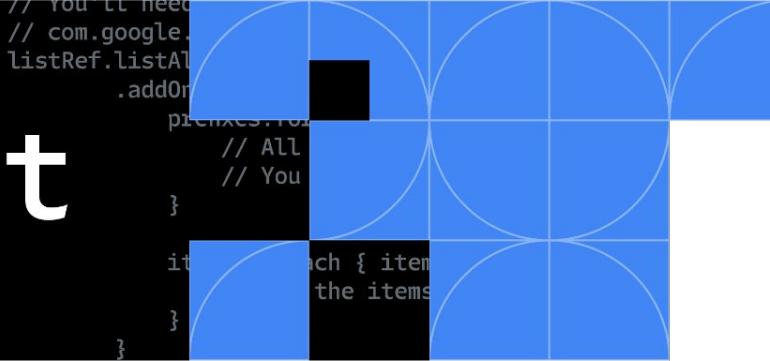
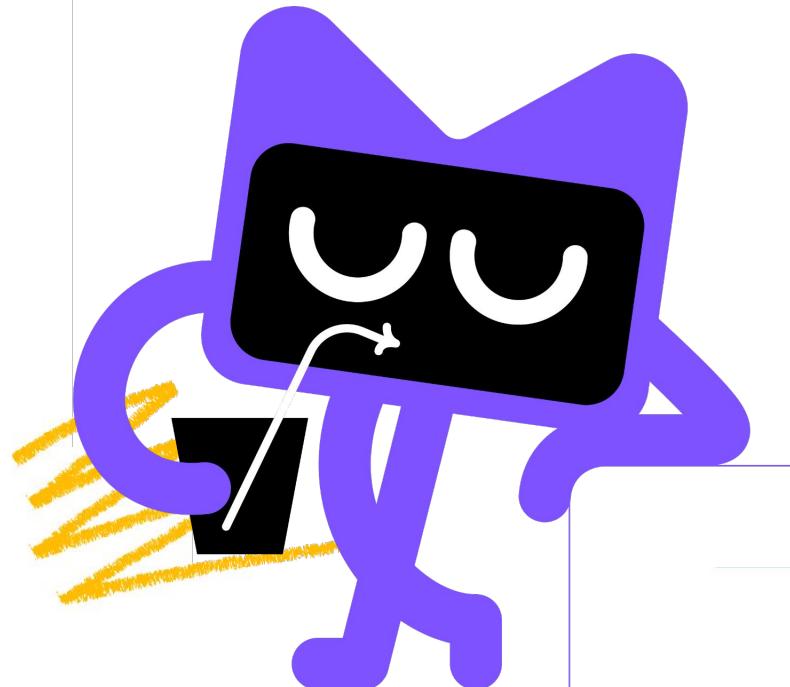


# devfest



## Kotlin Multiplatform (2024 Edition)

 Google Developer Groups  
Dublin, Ireland



# Questions we will answer

What is Kotlin Multiplatform / KMP?

To get started, how do I...

- Set up my machine?
- Create a KMP project?

Then how do I build an app:

- With a native UI?
- With a shared UI?

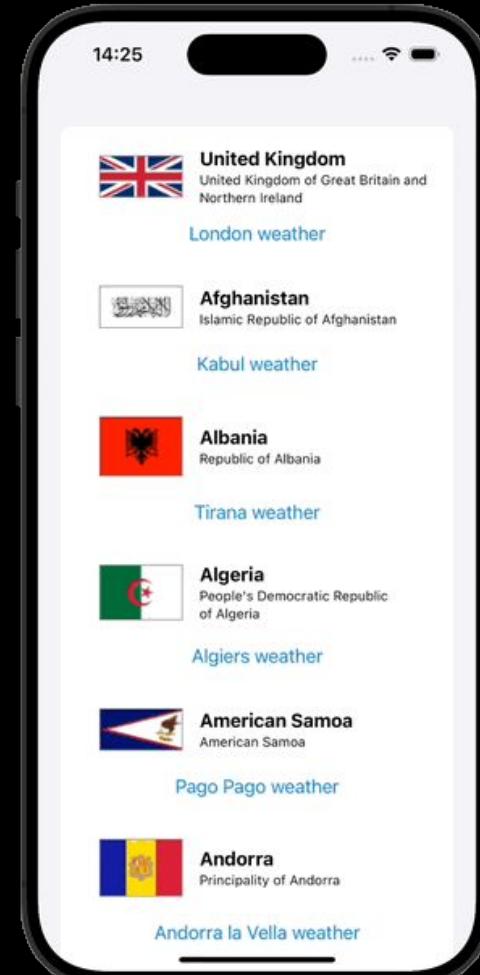
# The Weather App (Native UI)



Jetpack  
Compose



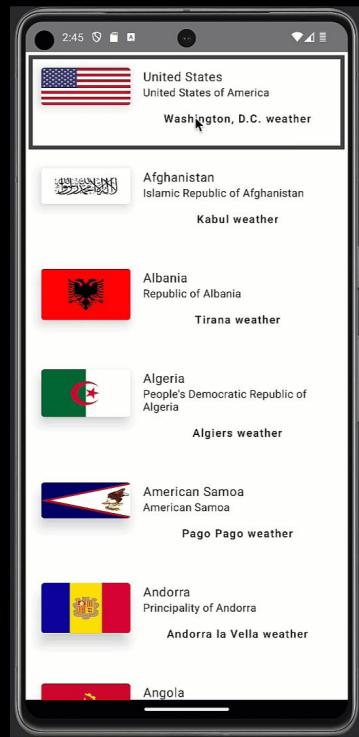
SwiftUI



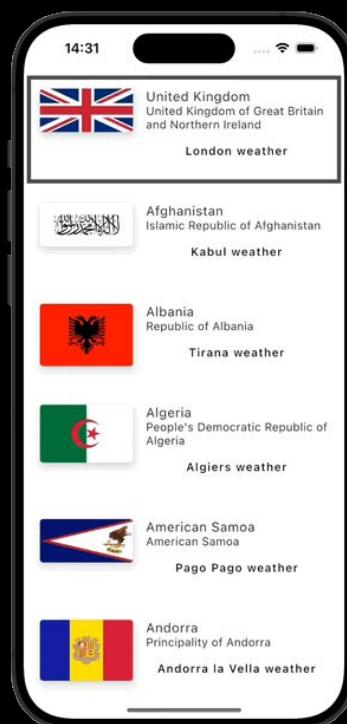
# The Weather App (Shared UI)



Android + CM



iOS + CM



Desktop + CM



## Access to resources

Sample code is in the repo below

Slides are included in the project as PDF

<https://github.com/garthgilmourni/devfest-ireland-2023>



November 21, 2023  
17:00 UTC

## The State of Kotlin Multiplatform

Svetlana Isakova  
Márton Braun



November 28, 2023  
16:00 UTC

## Build Apps for iOS, Android, and Desktop in 100% Kotlin with Compose Multiplatform

Sebastian Aigner  
Márton Braun



November 23, 2023  
16:00 UTC

## Build Apps for iOS and Android with Shared Logic and Native UIs

Pamela Hill  
Garth Gilmour



November 30, 2023  
16:00 UTC

## iOS Development with Kotlin Multiplatform: Tips and Tricks

Pamela Hill  
Tadeas Kriz



Kotlin by JetBrains 🎉 📺 @kotlin · Nov 1

## 🚀 Kotlin Multiplatform is Stable in Kotlin 1.9.20 and production-ready!

Learn about the evolution of KMP and what the Stable version brings.  
Discover how it can streamline your development process, and explore  
new learning resources to get started quickly:

# Kotlin Multiplatform Is Stable. Start Using It Now!

[blog.jetbrains.com](https://blog.jetbrains.com)



Kotlin by JetBrains 🎉 📺  
@kotlin

👉 We're working on adding lots of exciting things to Kotlin Multiplatform in 2024:

- ✓ Direct Kotlin-to-Swift export
- ✓ Compose for iOS in Beta
- ✓ A single IDE experience with Fleet
- ✓ Improved KMP library publishing process

Explore our roadmap for more [⬇️ blog.jetbrains.com/kotlin/2023/11...](https://blog.jetbrains.com/kotlin/2023/11...)

JET  
BRAINS



## Kotlin Multiplatform Development Roadmap for 2024

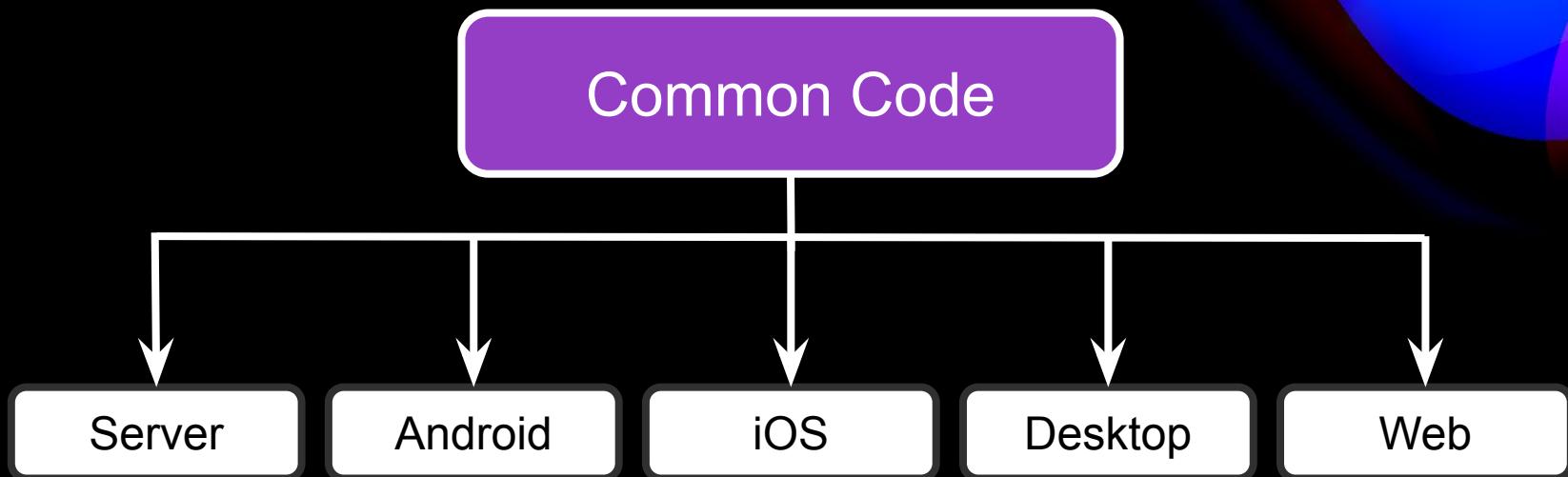
[blog.jetbrains.com](https://blog.jetbrains.com)

# What is Kotlin Multiplatform? (aka. KMP)

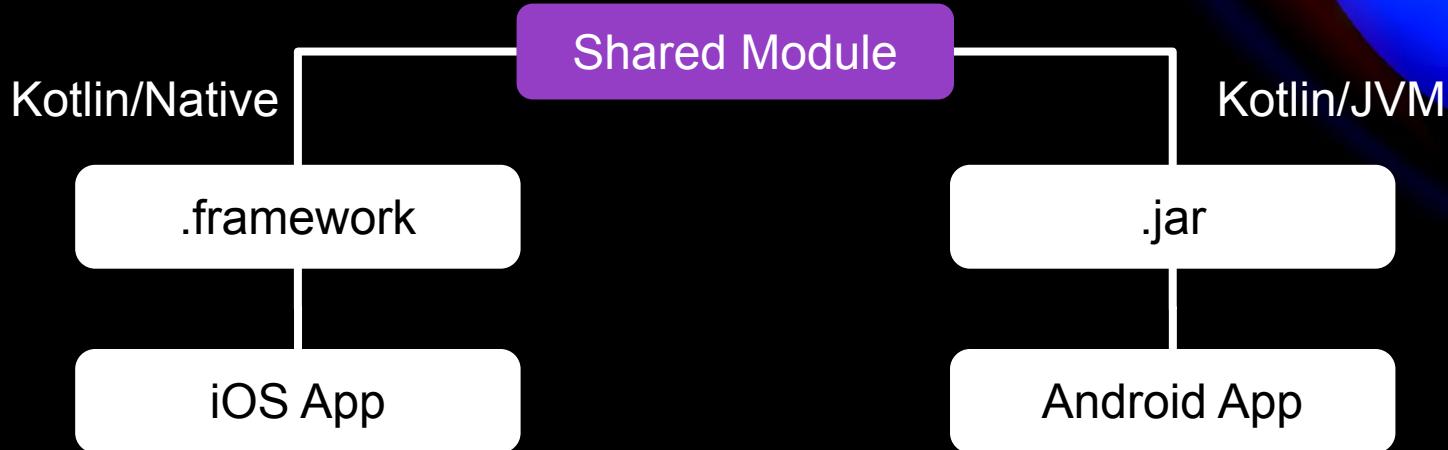


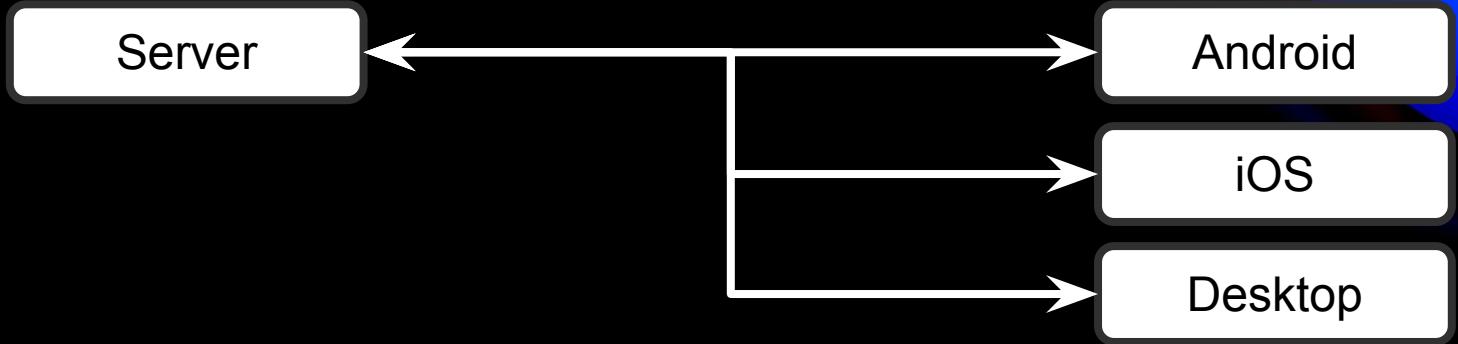
# Multiplatform Means Sharing

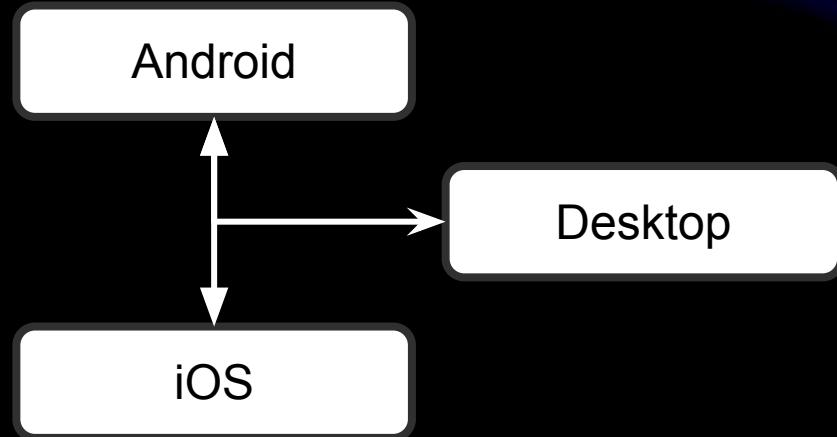
# Sharing code across platforms



# Sharing code across platforms



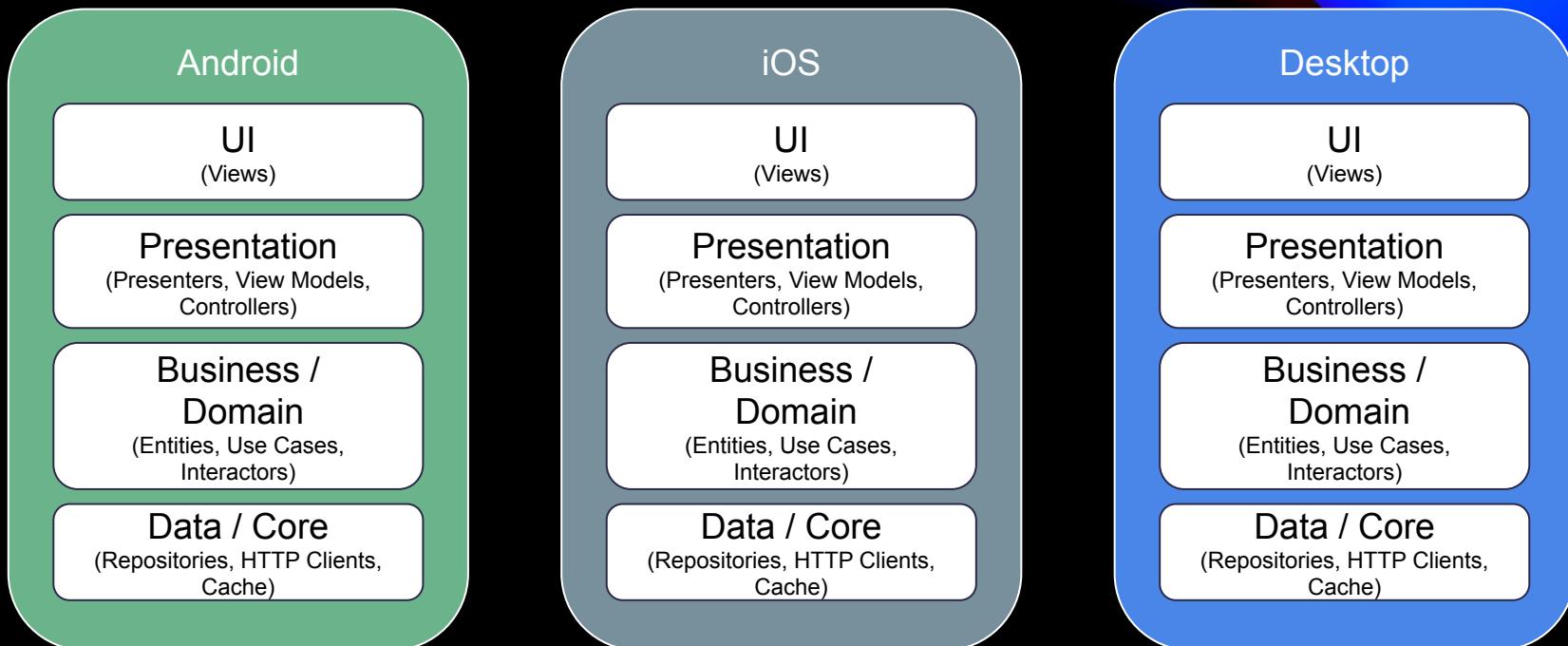




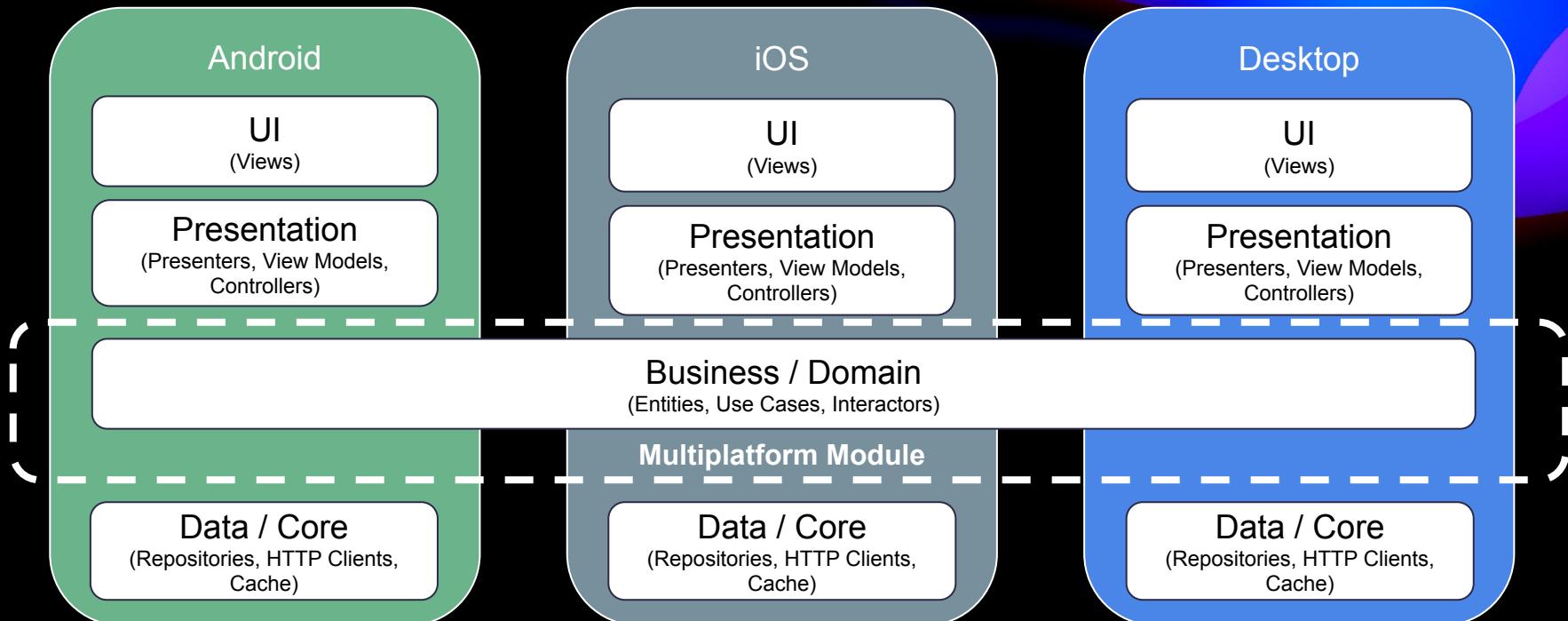


# What Do I Share?

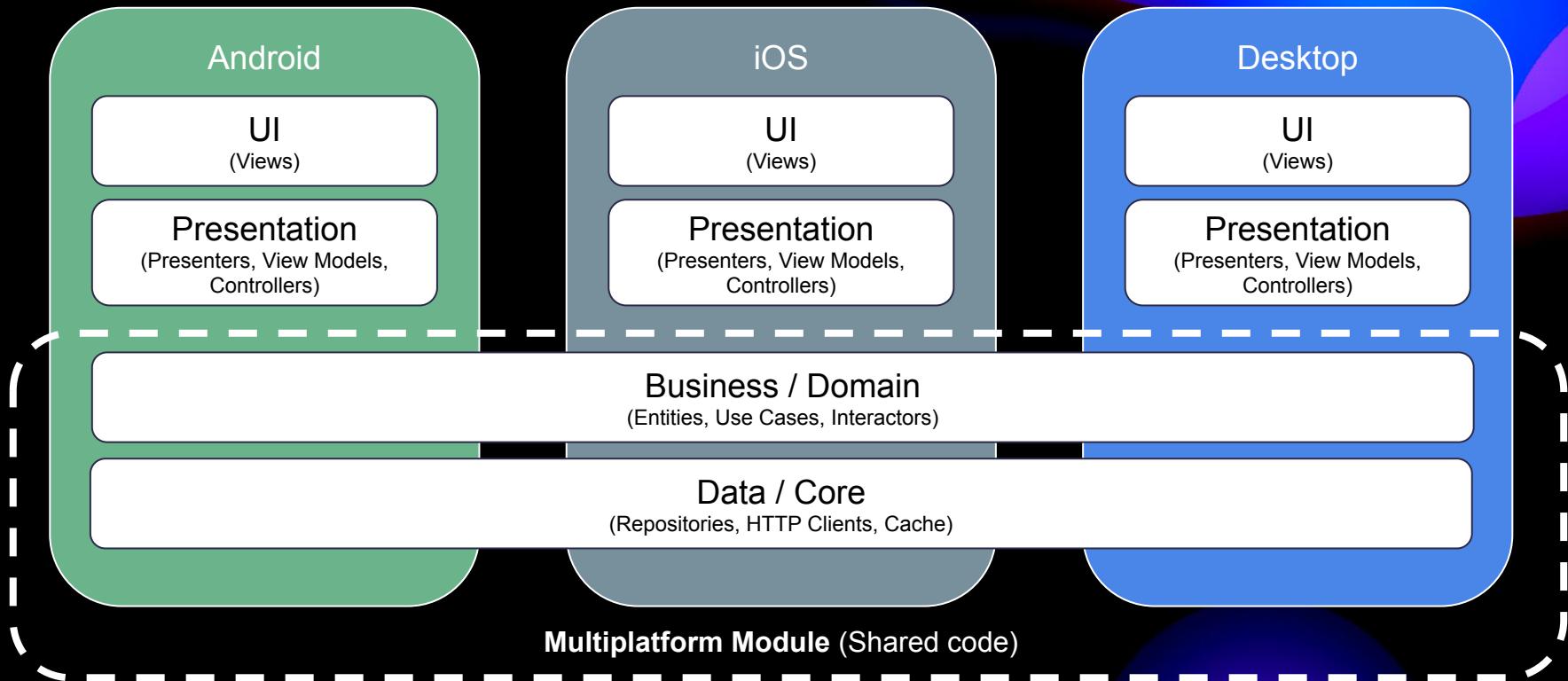
# Without KMP - separate stacks



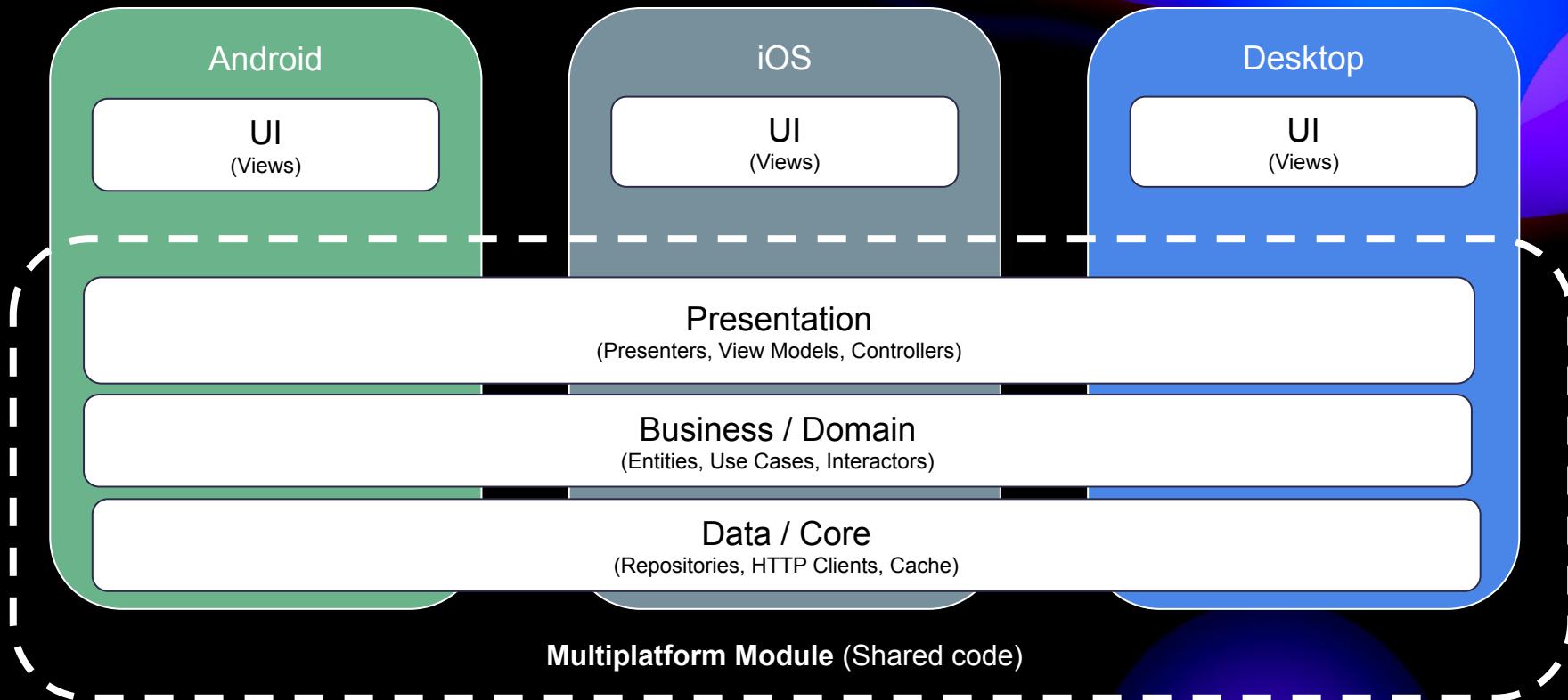
# With KMP - sharing logic & data



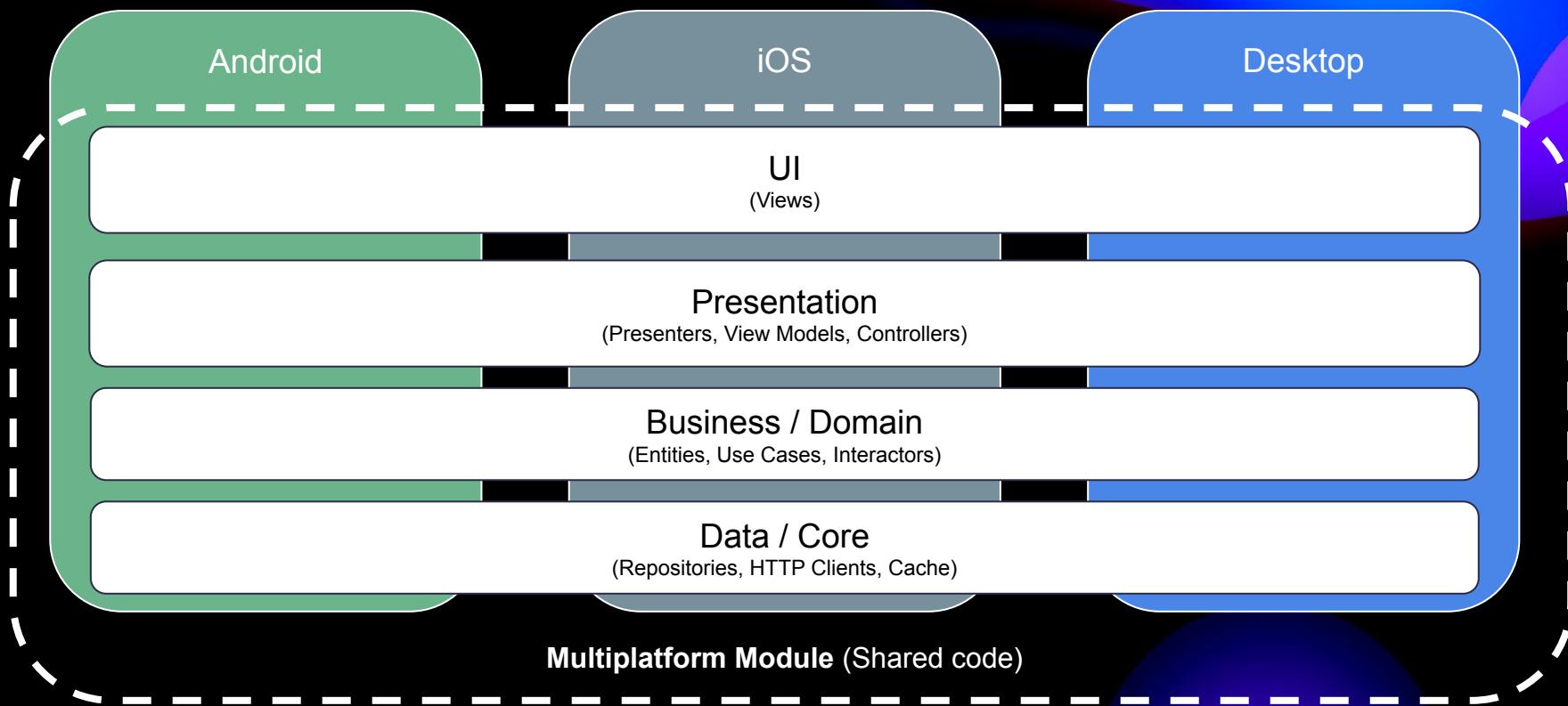
# With KMP - sharing logic, data and services



# With KMP - sharing logic, data, services & presentation



# With Compose Multiplatform - sharing everything



# Who Is Using KMP?



# NETFLIX



# PHILIPS vmware®

AUTODESK

chalk

m · meetup

# Forbes

io

# Define Stable?



# Strict compatibility guarantees

Compiler  
Support

Language  
Features

Library  
APIs

Build  
Tooling

IDE  
Support

# Core KMP stability

<b>Android</b>	Stable
<b>iOS</b>	Stable
<b>Desktop (JVM)</b>	Stable
<b>Server-side (JVM)</b>	Stable
<b>Web (Kotlin/Wasm)</b>	Experimental
<b>Web (Kotlin/JS)</b>	Stable
<b>watchOS</b>	Best effort
<b>tvOS</b>	Best effort

# Compose Multiplatform stability

<b>Android</b>	Stable
<b>iOS</b>	Alpha
<b>Desktop (JVM)</b>	Stable
<b>Web (Kotlin/Wasm)</b>	Experimental



# How Do I... Set Up My Machine?

# Set up an environment

 [Edit page](#) Last modified: 01 November 2023

This is the first part of the **Getting started with Kotlin Multiplatform** tutorial:

- 1 Set up an environment
- 2 Create your first cross-platform app
- 3 Update the user interface
- 4 Add dependencies
- 5 Share more logic
- 6 Wrap up your project

Before you create your first application that works on both iOS and Android, you'll need to set up an environment for Kotlin Multiplatform development.



To write iOS-specific code and run an iOS application on a simulated or real device, you'll need a Mac with macOS. This cannot be performed on other operating systems, such as Microsoft Windows. This is an Apple requirement.

To target Android

Install Android Studio

Create a Virtual Device

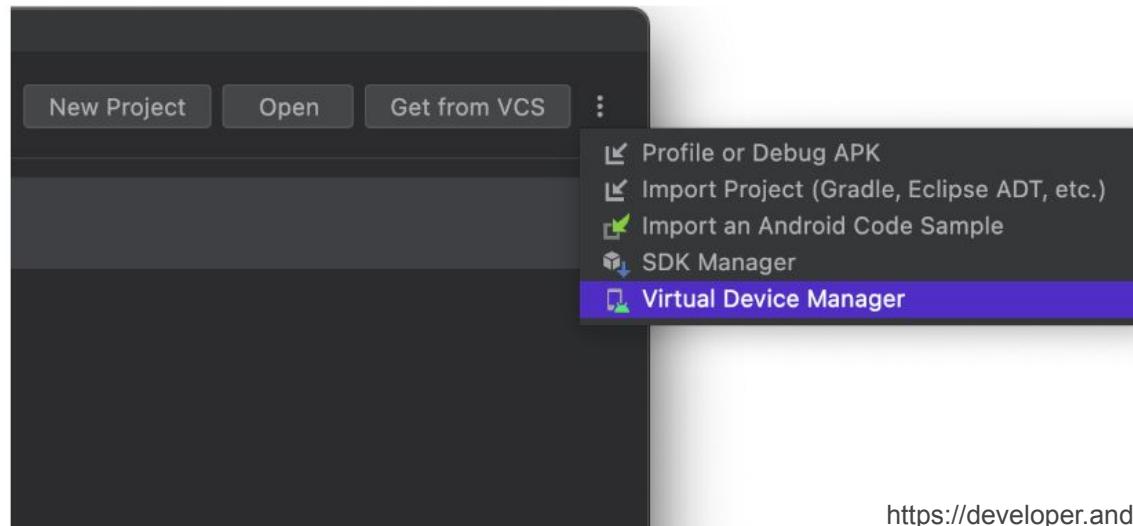
Ignore it

# Create and manage virtual devices

An Android Virtual Device (AVD) is a configuration that defines the characteristics of an Android phone, tablet, Wear OS, Android TV, or Automotive OS device that you want to simulate in the [Android Emulator](#). The Device Manager is a tool you can launch from Android Studio that helps you create and manage AVDs.

To open the new **Device Manager**, do one of the following:

- From the Android Studio Welcome screen, select **More Actions > Virtual Device Manager**.



# To target iOS

Install Xcode

Launch Xcode

Accept license terms

Restart on updates

Otherwise ignore it

# What about CocoaPods?

A dependency manager for Swift and Objective-C

You can configure it within your Gradle build file

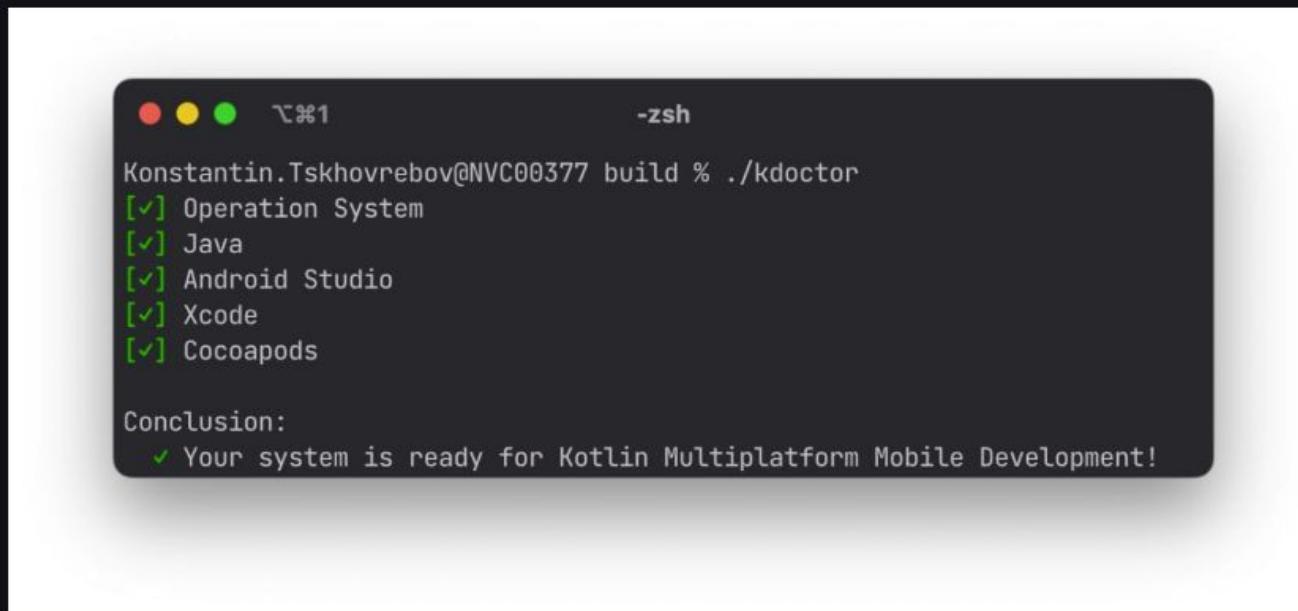
It can be used to declare native dependencies

It's up to you if you want to use CocoaPods

# KDoctor

JetBrains incubator license Apache License 2.0 homebrew v1.1.0

KDoctor is a command-line tool that helps to set up the environment for [Kotlin Multiplatform Mobile](#) app development.



A screenshot of a terminal window titled "-zsh". The window shows the output of the KDoctor command. It lists several dependencies with green checkmarks: Operation System, Java, Android Studio, Xcode, and Cocoapods. Below this, it says "Conclusion:" followed by a green checkmark and the message "Your system is ready for Kotlin Multiplatform Mobile Development!".

```
Konstantin.Tskhovrebov@NVC00377 build % ./kdoctor
[✓] Operation System
[✓] Java
[✓] Android Studio
[✓] Xcode
[✓] Cocoapods

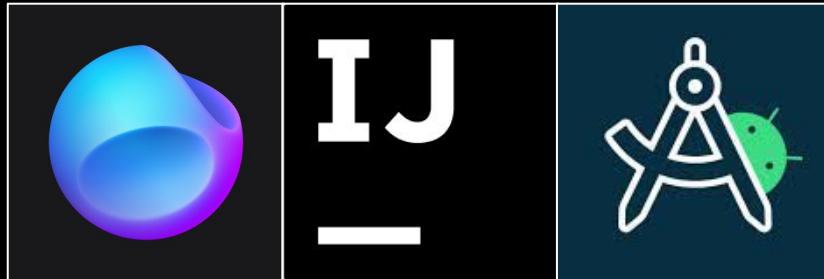
Conclusion:
✓ Your system is ready for Kotlin Multiplatform Mobile Development!
```

# Selecting your IDE

Android Studio will work well

IntelliJ IDEA can also be used

But do consider Fleet!



# Pros and cons of Fleet

**Pro:** Code completion and refactoring for Swift

**Pro:** Cross language navigation and refactoring (Swift  $\longleftrightarrow$  Kotlin)

**Pro:** Cross language debugging (Swift  $\longleftrightarrow$  Kotlin)

**Con:** Still in Public Preview so expect issues



# How Do I... Create a Project?

[kmp.jetbrains.com](http://kmp.jetbrains.com)

**Kotlin Multiplatform Wizard**

New Project    Template Gallery (Coming soon)     

Project Name: WebinarProject

Project ID: com.kmp.webinar

**Android**  
With Compose Multiplatform UI framework based on Jetpack Compose

**iOS**  
UI Implementation  
 Share UI (with Compose Multiplatform UI framework) (Alpha)  
 Do not share UI (use only SwiftUI)

**Desktop**

**Web** (Coming soon)  
Kotlin/Wasm is maturing rapidly. Soon, we'll enable Web project creation with Compose for Web in the wizard. In the meantime, check out these examples on GitHub for how to use Kotlin/Wasm.

**Server**

**DOWNLOAD**

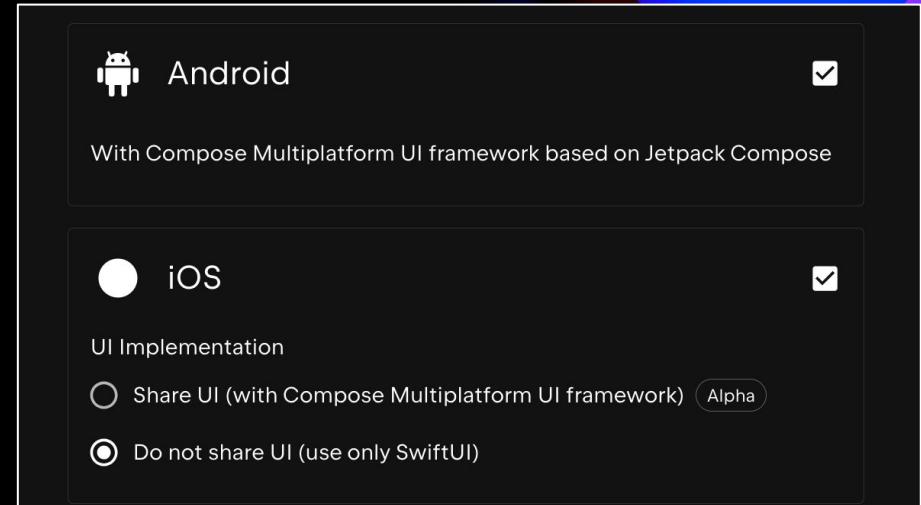
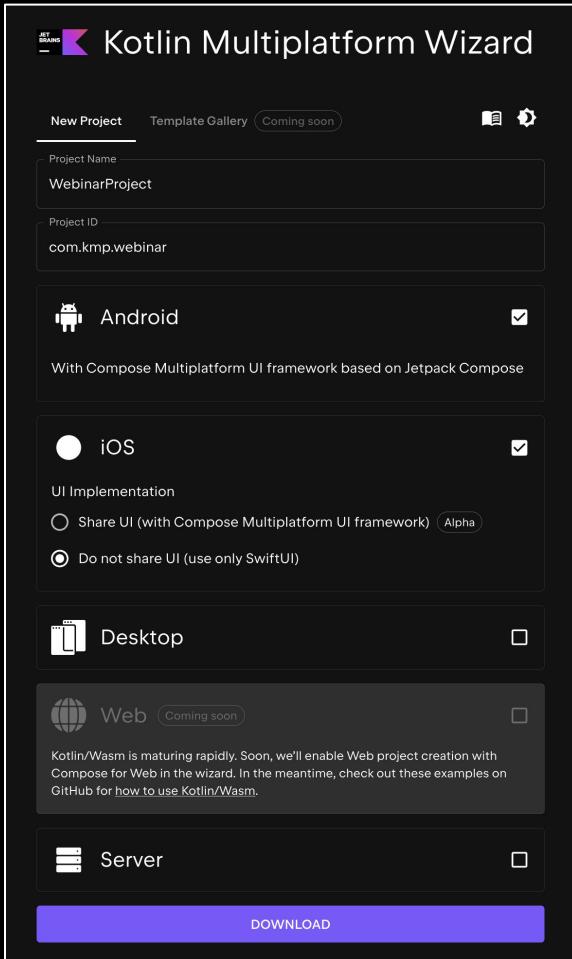


**Kotlin Multiplatform Wizard**

New Project    Template Gallery (Coming soon)     

Project Name: WebinarProject

Project ID: com.kmp.webinar



WebinarProject

> .fleet

> .gradle

> .idea

> composeApp

> gradle

> iosApp

✓ shared

> build

✓ src

> androidMain / kotlin

> commonMain / kotlin

> iosMain / kotlin

build.gradle.kts

.gitignore

build.gradle.kts

gradle.properties

gradlew

gradlew.bat

local.properties

README.md

settings.gradle.kts

> External Libraries

build.gradle.kts

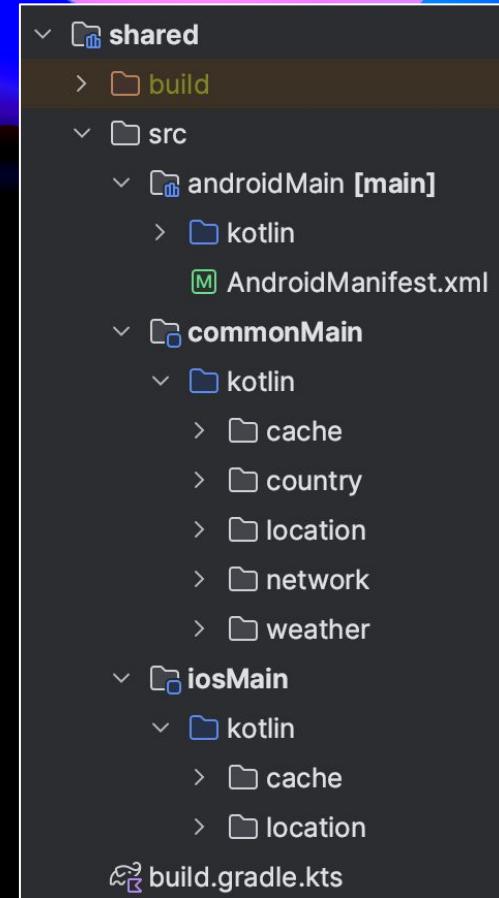
```
1  plugins { this: PluginDependenciesSpecScope
2      alias(libs.plugins.kotlinMultiplatform)
3      alias(libs.plugins.androidLibrary)
4  }
5
6  kotlin { this: KotlinMultiplatformExtension
7      listOf(
8          iosX64(),
9          iosArm64(),
10         iosSimulatorArm64()
11     ).forEach { iosTarget: KotlinNativeTarget →
12         iosTarget.binaries.framework { this: Framework
13             baseName = "Shared"
14             isStatic = true
15         }
16     }
17
18     androidTarget { this: KotlinAndroidTarget
19         compilations.all { this: KotlinJvmAndroidCompilation
20             kotlinOptions { this: KotlinJvmOptions
21                 jvmTarget = "1.8"
22             }
23         }
24     }
25 }
```

shared / build.gradle.kts

# The shared module

~~This only holds code which  
works on all platforms~~

This is where we develop the code  
that will be shared across platforms

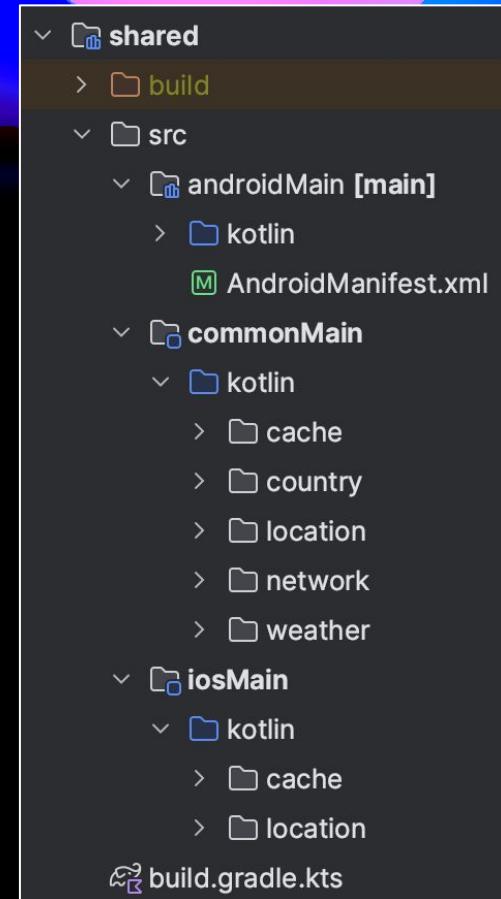


# The **shared** module

This is a Kotlin Multiplatform Module

It contains 3 source sets

- commonMain
- androidMain
- iosMain

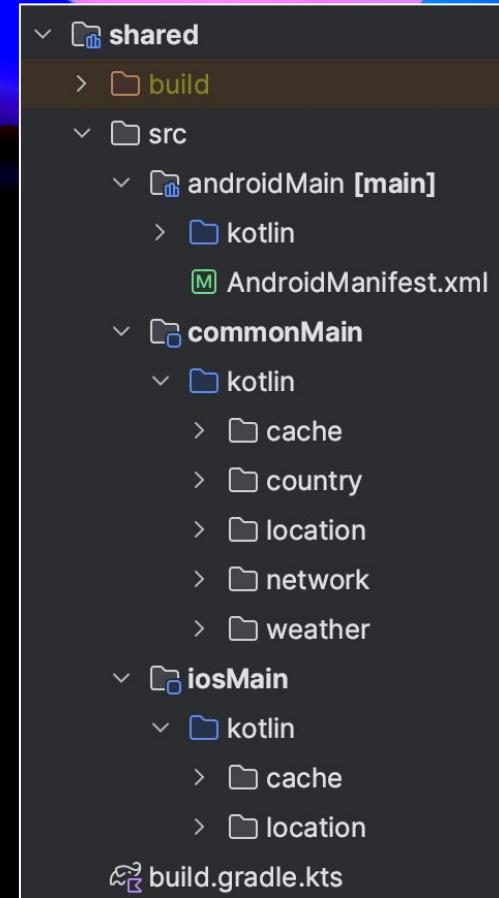


# The shared module

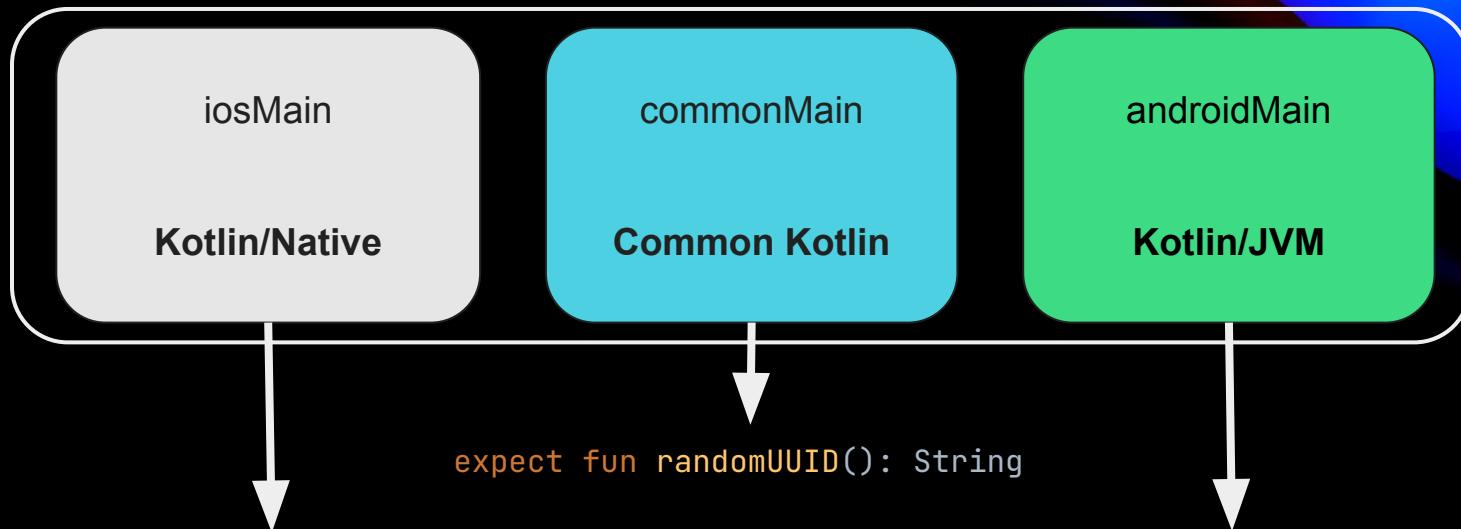
The source sets are compiled in combination:

commonMain + androidMain = 

commonMain + iosMain = 



# Introducing expect / actual functions



```
import platform.Foundation.NSUUID
actual fun randomUUID(): String =
    NSUUID().UUIDString()
```

```
expect fun randomUUID(): String
```

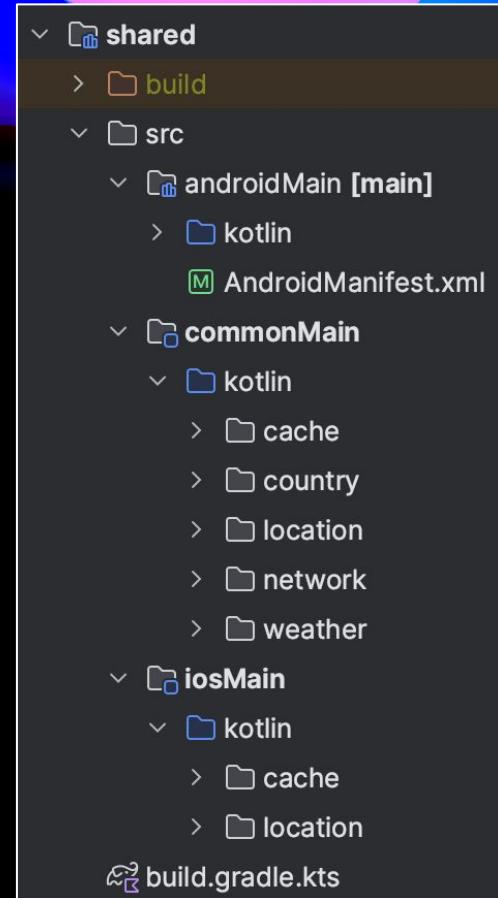
```
import java.util.*
actual fun randomUUID() =
    UUID.randomUUID().toString()
```

# The shared module (summary)

commonMain contains common code

The only dependencies will be on  
multiplatform libraries (like KStore)

Expected declarations need matching actual  
declarations in platform specific source sets



# Advice on expect / actual functions

A few expected declarations are fine

Lots of them could be a code smell

- Create interfaces to model abstractions
- Use expected functions as factories
- Consider adopting a DI framework

```
interface Platform {  
    val name: String  
}  
  
expect fun getPlatform(): Platform
```

```
class AndroidPlatform: Platform {  
    override val name: String =  
        "Android ${Build.VERSION.SDK_INT}"  
}  
  
actual fun getPlatform() = AndroidPlatform()
```

```
class iOSPlatform: Platform {  
    override val name: String =  
        UIDevice.currentDevice.systemName()  
        + " "  
        + UIDevice.currentDevice.systemVersion  
}  
  
actual fun getPlatform() = iOSPlatform()
```

# The composeApp module

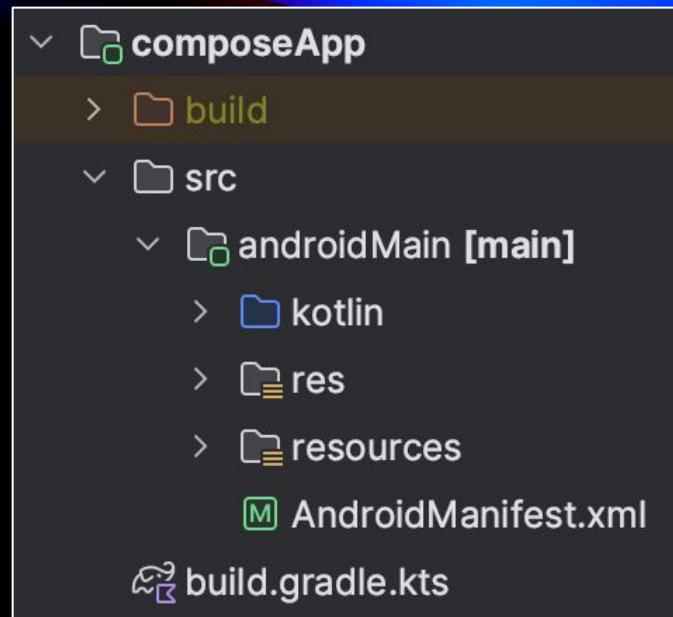
This is a Kotlin Module

It contains a single source set

- In the Native UI use case

This source set holds

- Your Jetpack Compose based UI
- Other Android types (e.g. Activities)



# The iosApp Folder

This is an Xcode project

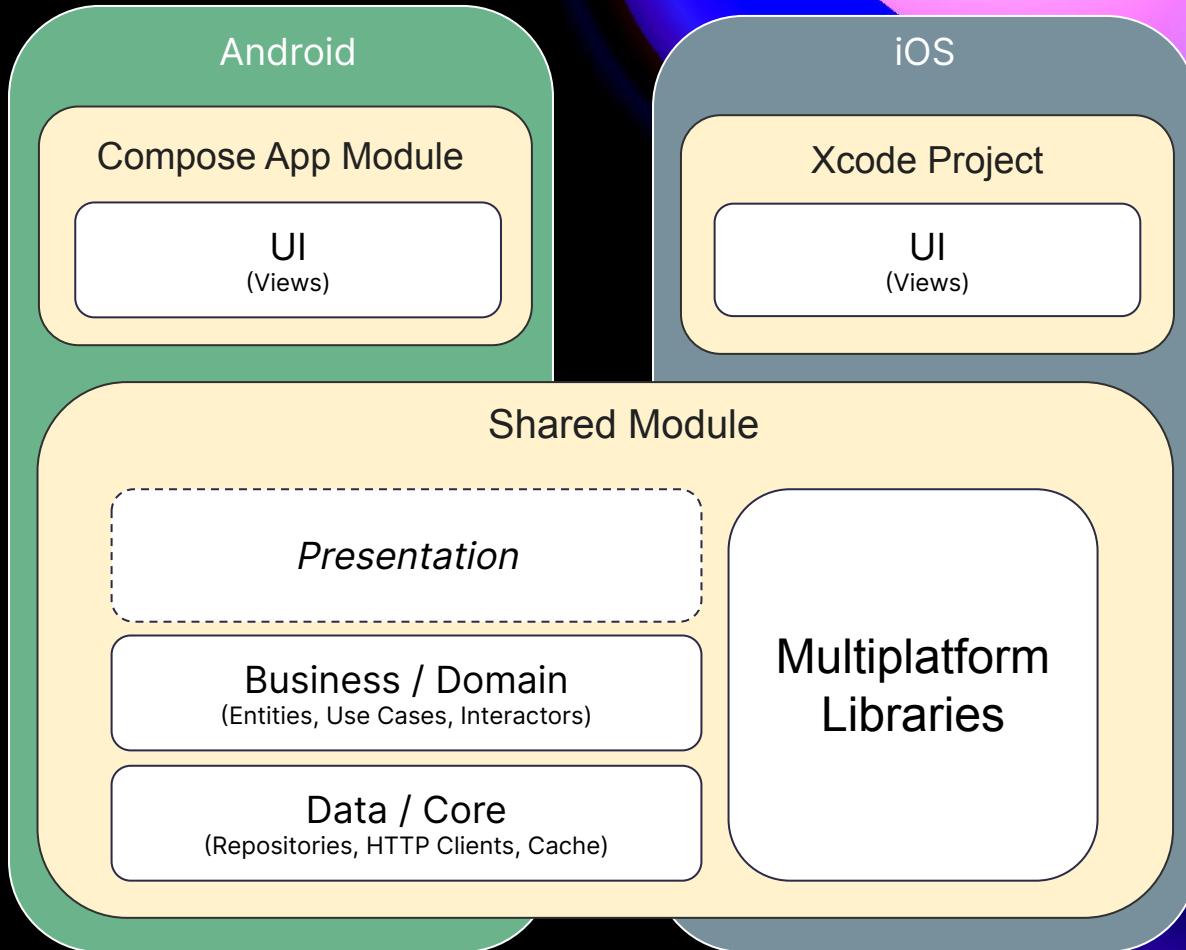
Containing the infrastructure needed  
to run your application on iOS

This is where we place Swift code

In this case our Native UI

```
< iosApp
  > Configuration
< iosApp
  > Assets.xcassets
  > Preview Content / Preview Assets.xcassets
  < ContentView.swift
  < CountriesView.swift
  < Country.swift
  < CountryDetailsView.swift
  < CountryRowView.swift
  <> Info.plist
  < iOSApp.swift
  < WeatherView.swift
> iosApp.xcodeproj
```

# Building an App (with Native UI)



A screenshot of a web browser window showing a GitHub repository page for "terrakok/kmp-awesome". The URL in the address bar is <https://github.com/terrakok/kmp-awesome>. The page title is "GitHub - terrakok/kmp-awesome". The browser interface includes standard navigation buttons (back, forward, search) and a tab bar.

## ☰ README.MD

# Awesome Kotlin Multiplatform



PRs welcome · awesome · stars 2k · maven-central · v2.0.0-Beta1

Kotlin Multiplatform technology simplifies the development of cross-platform projects. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

This list contains libraries which support iOS and Android targets in first place.

## Resources

- [Website](#)
- [Web Wizard](#)
- [Compose Multiplatform Wizard](#)
- [Documentation](#)
- [Blog](#)
- [YouTube](#)
- [Samples](#)
- [Jetpack Compose Components](#)
- [Kotlin Multiplatform by Tutorials](#)
- [Simplifying Application Development with Kotlin Multiplatform Mobile](#)

Readme

Activity

2k stars

48 watching

113 forks

Report repository

## Releases 11

Issue 11 · Latest  
on Sep 18

+ 10 releases

## Contributors 48



+ 37 contributors

# We will use KStore

The screenshot shows the GitHub page for the KStore library. At the top, there's a purple header with the KStore logo and a link to the repository. Below the header, a green button indicates a "Build passing". The version is listed as "alpha" (pink), "Kotlin 1.8.21" (purple), "maven-central" (blue), and "v0.6.0" (blue). A horizontal bar below shows supported platforms: "platform android" (green), "platform ios" (light gray), "platform macos" (light gray), "platform watchos" (light gray), "platform tvos" (light gray), "platform jvm" (red), "platform linux" (purple), "platform windows" (purple), "platform jsNode" (yellow), and "platform jsBrowser" (yellow). A descriptive paragraph explains the library is a tiny Kotlin multiplatform library for saving and restoring objects using kotlinx.coroutines, kotlinx.serialization, and okio, inspired by RxStore. The "Features" section lists several bullet points with icons: a lock for read-write locks, a memory icon for in-memory caching, a clipboard icon for default values, a car icon for migration support, and a laptop icon for multiplatform support.

**K Store**

Build **passing**

alpha **Kotlin 1.8.21** maven-central **v0.6.0**

platform **android** platform **ios** platform **macos** platform **watchos** platform **tvos** platform **jvm** platform **linux** platform **windows**  
platform **jsNode** platform **jsBrowser**

A tiny Kotlin multiplatform library that assists in saving and restoring objects to and from disk using `kotlinx.coroutines`, `kotlinx.serialization` and `okio`. Inspired by [RxStore](#)

## Features

- Read-write locks; with a mutex FIFO lock
- In-memory caching; read once from disk and reuse
- Default values; no file? no problem!
- Migration support; moving shop? take your data with you
- Multiplatform!

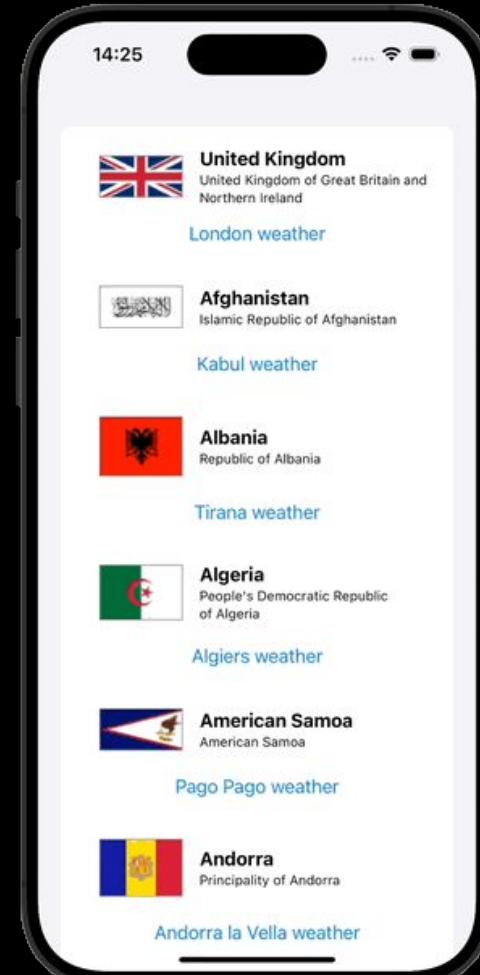
<https://github.com/xxfast/KStore>



Jetpack  
Compose



SwiftUI



# What do we need?

Domain types

Networking code

Support for caching

Platform specific support:

- For creating the cache file
- For working with locations

A Jetpack Compose based interface

A SwiftUI based interface

Data in interfaces

# What do we need?

Domain types

Networking code

Support for caching

Platform specific support:

- For creating the cache file
- For working with locations

A JetPack Compose based interface

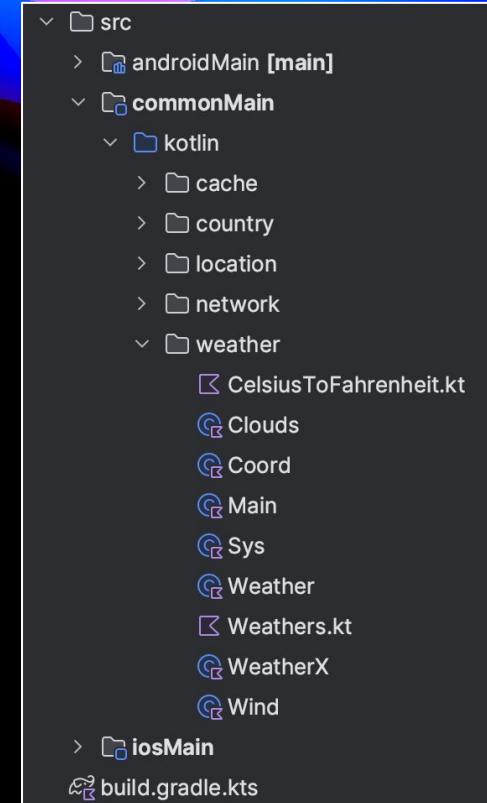
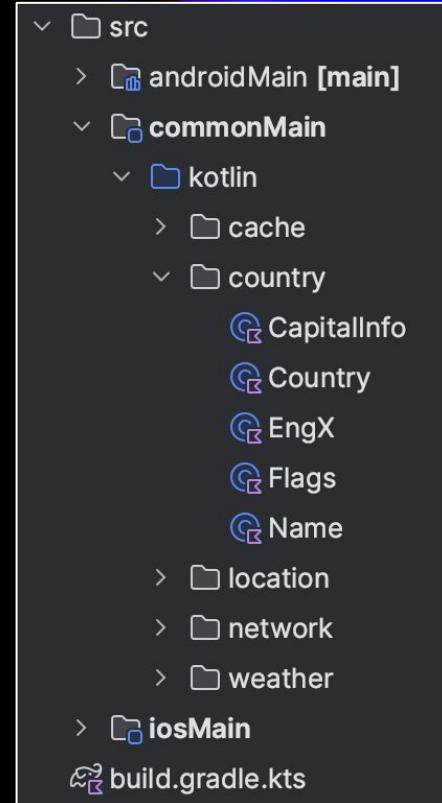
A SwiftUI based interface

Data in interfaces

# Our domain types

We have two subdomains:

- One to model countries
- Another to model weather



# What do we need?

Domain types ✓

Networking code

Support for caching

Platform specific support:

- For creating the cache file
- For working with locations

A JetPack Compose based interface

A SwiftUI based interface

Data in interfaces

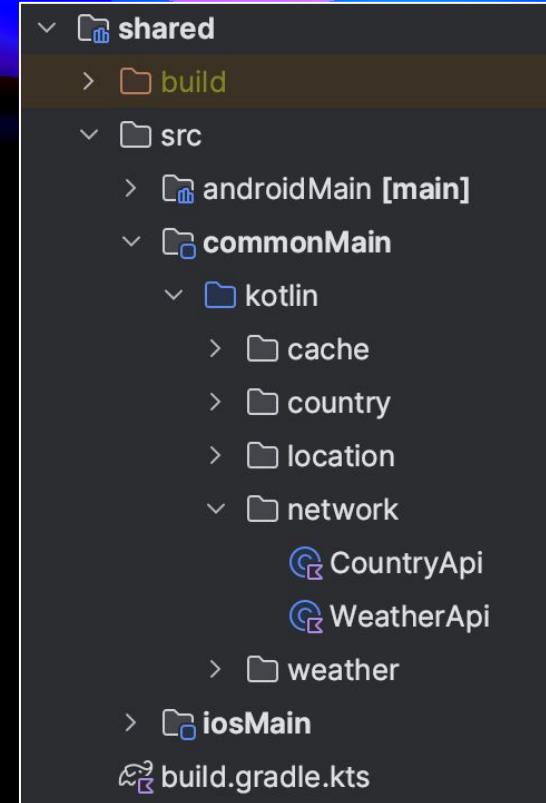
# Types to support networking

Our networking code uses two servers:

- [restcountries.com](https://restcountries.com) for countries
- [api.openweathermap.org](https://api.openweathermap.org) for weather

We have a client for each one:

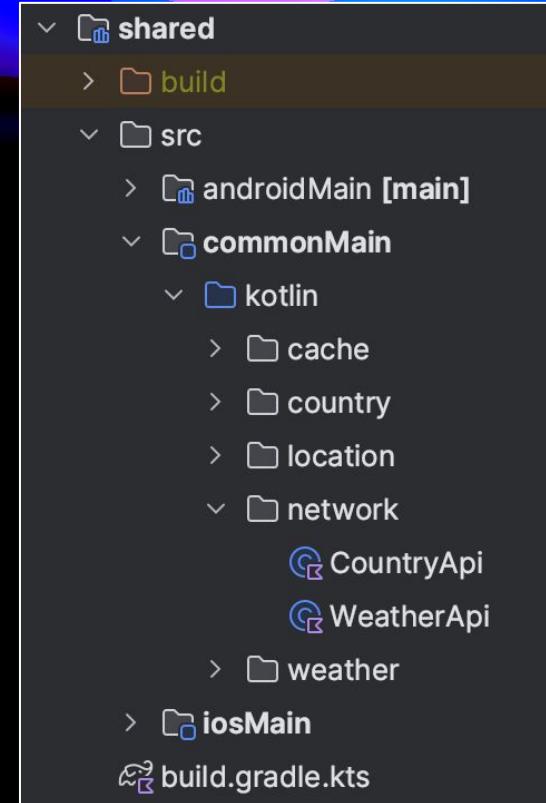
- CountryApi
- WeatherApi



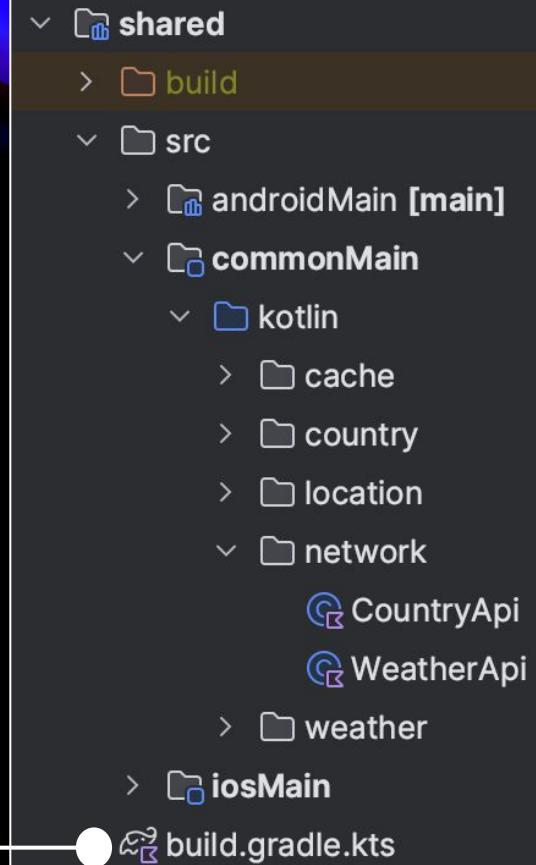
# Types to support networking

Multiplatform libraries handle the heavy lifting:

- Ktor Client to send the requests
- Kotlinx Serialization for marshalling
- KStore to cache the results we obtain



```
sourceSets {  
    all {  
        ...  
    }  
  
    commonMain.dependencies {  
        ...  
    }  
  
    androidMain.dependencies {  
        ...  
    }  
  
    iosMain.dependencies {  
        ...  
    }  
}
```



```
shared  
└── build  
shared  
└── src  
    ├── androidMain [main]  
    └── commonMain  
        ├── kotlin  
        ├── cache  
        ├── country  
        ├── location  
        └── network  
            └── CountryApi  
            └── WeatherApi  
        └── weather  
    └── iosMain  
build.gradle.kts
```

```
sourceSets {  
    all {  
        languageSettings.optIn("kotlin.experimental.ExperimentalObjCName")  
    }  
  
    commonMain.dependencies {  
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")  
  
        implementation("io.ktor:ktor-client-core:2.3.3")  
        implementation("io.ktor:ktor-client-content-negotiation:2.3.3")  
        implementation("io.ktor:ktor-serialization-kotlinx-json:2.3.3")  
  
        implementation("io.github.xxfast:kstore:0.6.0")  
        implementation("io.github.xxfast:kstore-file:0.6.0")  
    }  
  
    androidMain.dependencies {  
        implementation("io.ktor:ktor-client-android:2.3.3")  
    }  
  
    iosMain.dependencies {  
        implementation("io.ktor:ktor-client-darwin:2.3.3")  
    }  
}
```

```
suspend fun getAllCountries(): List<Country> {  
    return httpClient.get("https://restcountries.com/v3.1/all")  
        .body<List<Country>>()  
        .sortedBy { it.name.common }  
}
```

```
@Serializable  
data class Country(  
    val capital: List<String> = emptyList(),  
    val capitalInfo: CapitalInfo? = null,  
    val flags: Flags,  
    val name: Name,  
    val cca2: String  
)
```

```
suspend fun getWeather(lat: Double, long: Double): Weather {  
    val key = Config.WeatherApiKey  
    val URL = "https://api.openweathermap.org/data/2.5/weather"  
    val queryString = "?lat=$lat&lon=$long&appid=$key&units=metric"  
  
    return httpClient.get("$URL$queryString").body()  
}
```

```
@Serializable  
data class Weather(  
    val base: String,  
    val clouds: Clouds,  
    val cod: Int,  
    val coord: Coord,  
    val dt: Int,  
    val id: Int,  
    val main: Main,  
    val name: String,  
    val sys: Sys,  
    val timezone: Int,  
    val visibility: Int,  
    val weather: List<WeatherX>,  
    val wind: Wind  
)
```

# What do we need?

Domain types ✓

Networking code ✓

Support for caching

Platform specific support:

- For creating the cache file
- For working with locations

A JetPack Compose based interface

A SwiftUI based interface

Data in interfaces

# Using KStore for caching

The screenshot shows the GitHub page for the KStore library. At the top, there's a purple header with the KStore logo and a link to the repository. Below the header, a green button indicates a "Build passing". The version is listed as "alpha 1.8.21 maven-central v0.6.0". A horizontal bar below shows support for various platforms: android, ios, macos, watchos, tvos, jvm (highlighted in red), linux, windows, jsNode (highlighted in yellow), and jsBrowser. A descriptive paragraph explains the library's purpose: "A tiny Kotlin multiplatform library that assists in saving and restoring objects to and from disk using kotlinx.coroutines, kotlinx.serialization and okio. Inspired by [RxStore](#)". The "Features" section lists several key features with corresponding icons:

- Read-write locks; with a mutex FIFO lock
- In-memory caching; read once from disk and reuse
- Default values; no file? no problem!
- Migration support; moving shop? take your data with you
- Multiplatform!

<https://github.com/xxfast/KStore>

```
sourceSets {  
    all {  
        languageSettings.optIn("kotlin.experimental.ExperimentalObjCName")  
    }  
  
    commonMain.dependencies {  
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")  
  
        implementation("io.ktor:ktor-client-core:2.3.3")  
        implementation("io.ktor:ktor-client-content-negotiation:2.3.3")  
        implementation("io.ktor:ktor-serialization-kotlinx-json:2.3.3")  
        implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.4.0")  
  
        implementation("io.github.xxfast:kstore:0.6.0")  
        implementation("io.github.xxfast:kstore-file:0.6.0")  
    }  
  
    androidMain.dependencies {  
        implementation("androidx.startup:startup-runtime:1.2.0-alpha02")  
    }  
  
    iosMain.dependencies {  
        implementation("io.ktor:ktor-client-darwin:2.3.3")  
    }  
}
```

# Setting up the cache

```
// In shared/src/commonMain/kotlin/cache/CountrySDK.kt

class CountrySDK {

    private val cache: KStore<List<Country>>
        = storeOf(filePath = pathToCountryCache())

    ...
}
```

# Adding logic for caching

```
// In shared/src/commonMain/kotlin/cache/CountrySDK.kt

private suspend fun getSortedCountries(): List<Country> {
    return cache.get()
        ?: api.getAllCountries().also {
            cache.set(it)
        }
}
```

# What do we need?

Domain types ✓

Networking code ✓

Support for caching ✓

Platform specific support:

- For creating the cache file
- For working with locations

A JetPack Compose based interface

A SwiftUI based interface

Data in interfaces

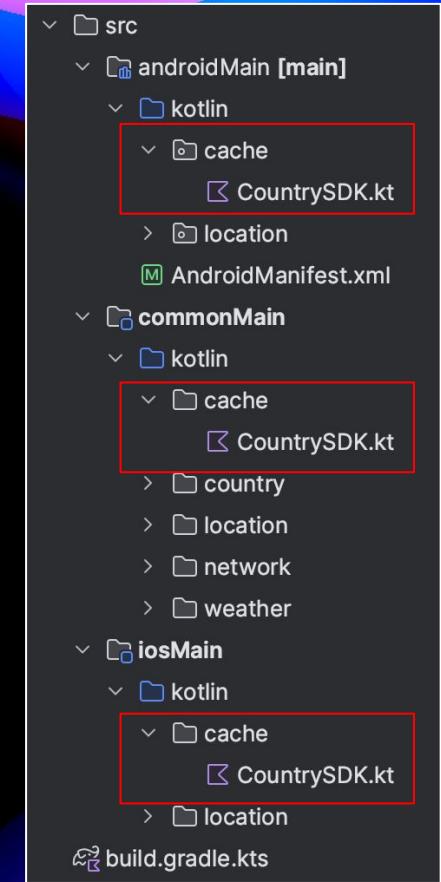
# Platform specific types for caching

Our KStore code requires a JSON file

How and where it is created is platform specific

So we expect a function in **commonMain**

Actual declarations go in **androidMain** and **iosMain**



```
// In shared/src/commonMain/kotlin/cache  
  
expect fun pathToCountryCache(): String
```

```
// In shared/src/androidMain/kotlin/cache  
  
lateinit var filePath: String  
  
actual fun pathToCountryCache(): String = filePath
```

```
// In composeApp/src/androidMain/kotlin  
  
class WebinarApplication : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
  
        filePath = "${filesDir.path}/country_cache.json"  
    }  
}
```



```
// In shared/src/commonMain/kotlin/cache  
  
expect fun pathToCountryCache(): String
```

```
// In shared/src/iosMain/kotlin/cache  
  
actual fun pathToCountryCache(): String  
    = "${NSHomeDirectory()}/country_cache.json"
```



# What do we need?

Domain types ✓

Networking code ✓

Support for caching ✓

Platform specific support:

- For creating the cache file ✓
- For working with locations

A JetPack Compose based interface

A SwiftUI based interface

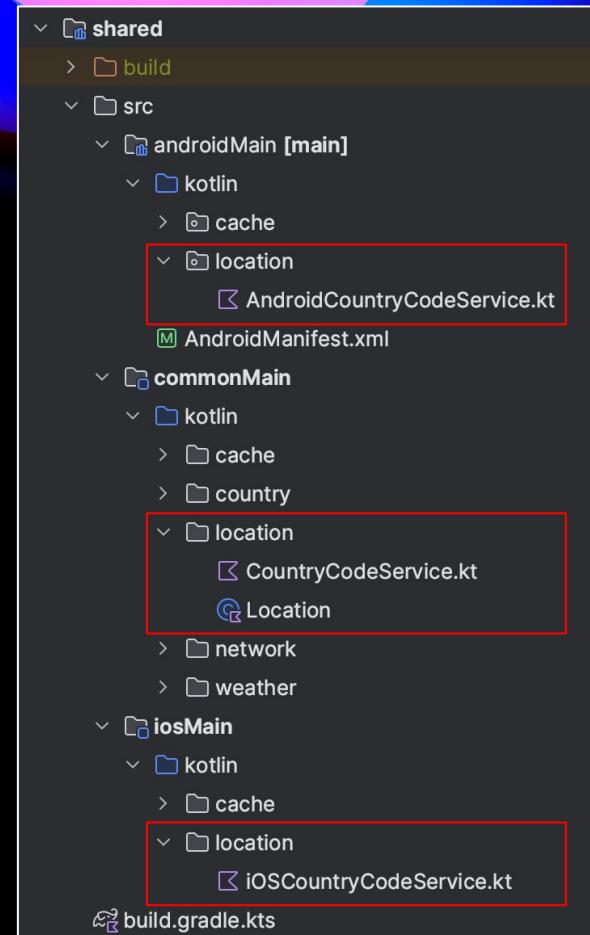
Data in interfaces

# Platform specific types for locations

We need to work with Country Codes

The way these are found is platform-specific

So once again we use **expect** and **actual**



```
// In shared/src/commonMain/kotlin/location

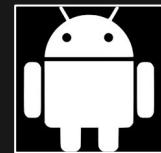
interface CountryCodeService {
    fun getCountryCode(): String?
}

expect fun getCountryCodeService(): CountryCodeService
```

```
// In shared/src/androidMain/kotlin/location

class AndroidCountryCodeService() : CountryCodeService {
    override fun getCountryCode(): String? {
        return Locale.getDefault().country
    }
}

actual fun getCountryCodeService(): CountryCodeService
    = AndroidCountryCodeService()
```



```
// In shared/src/commonMain/kotlin/location

interface CountryCodeService {
    fun getCountryCode(): String?
}
expect fun getCountryCodeService(): CountryCodeService
```

```
// In shared/src/iosMain/kotlin/location

class iOSCountryCodeService() : CountryCodeService {
    override fun getCountryCode(): String? {
        return NSLocale.currentLocale()
            .objectForKey(NSLocaleCountryCode)
            .toString()
    }
}

actual fun getCountryCodeService(): CountryCodeService
    = iOSCountryCodeService()
```



# Sorting logic

```
// In shared/src/commonMain/kotlin/cache/CountrySDK.kt

@NativeCoroutines
@Throws(Exception::class)
suspend fun getCountries(): List<Country> {
    val countryCode = getCountryCodeService().getCountryCode()

    val tempCountries = getSortedCountries().toMutableList()
    val currentCountry = tempCountries.first { it.cca2 == countryCode }
    tempCountries.remove(currentCountry)
    tempCountries.add(0, currentCountry)
    return tempCountries
}
```

# What do we need?

Domain types ✓

Networking code ✓

Support for caching ✓

Platform specific support: ✓

- For creating the cache file ✓
- For working with locations ✓

A JetPack Compose based interface

A SwiftUI based interface

Data in interfaces

# The Android UI: Jetpack Compose

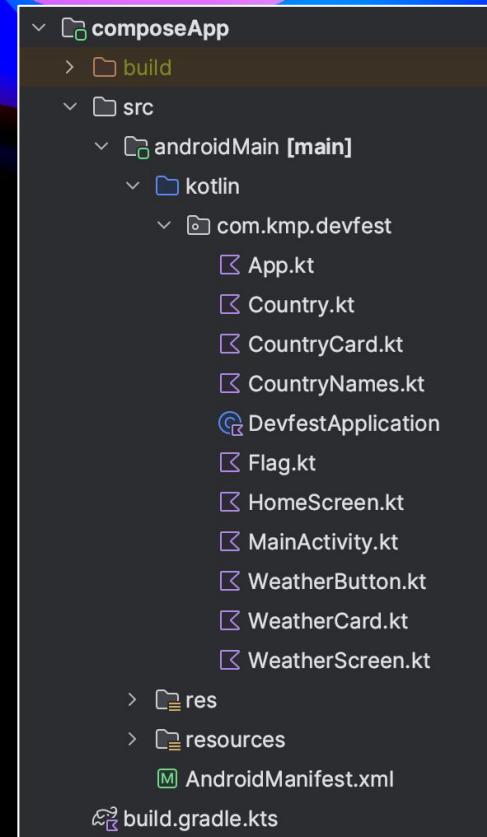
Our Android UI uses Jetpack Compose

- It is made up of Composable Functions

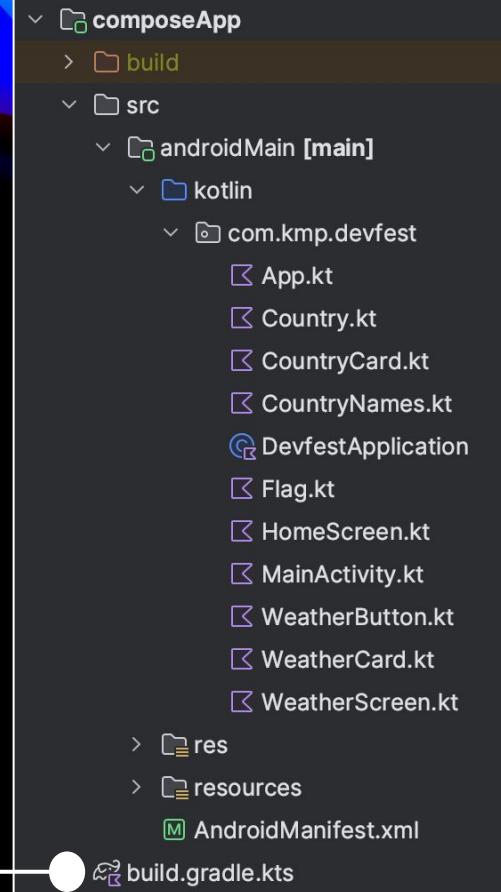
These use Android specific libraries

- E.g. the Coil library to display images

Hence they live in **androidMain**



```
sourceSets {  
  
    androidMain.dependencies {  
        implementation(libs.compose.ui)  
        implementation(libs.compose.ui.tooling.preview)  
        implementation(libs.androidx.activity.compose)  
  
        implementation("io.coil-kt:coil-compose:2.5.0")  
    }  
    commonMain.dependencies {  
        ...  
    }  
}
```



# The Country Composable

CountryNames  
Composable



Flag  
Composable



Germany  
Federal Republic of Germany

Berlin weather

WeatherButton  
Composable

```
@Composable
fun Country(modifier: Modifier, country: Country) {
    Row(modifier = Modifier.padding(8.dp)) {
        Column(modifier = Modifier.width(130.dp)) {
            Flag(
                modifier = Modifier.fillMaxWidth().padding(8.dp),
                Country.flags
            )
        }
        Column(modifier = Modifier.fillMaxWidth().padding(8.dp)) {
            CountryNames(name = country.name)
            val capitalInfo = country.capitalInfo
            if (country.capital.isNotEmpty() && capitalInfo != null) {
                WeatherButton(
                    capitals = country.capital,
                    capitalInfo = capitalInfo
                )
            }
        }
    }
}
```

# What do we need?

Domain types ✓

Networking code ✓

Support for caching ✓

Platform specific support:

- For creating the cache file ✓
- For working with locations ✓

A JetPack Compose based interface ✓

A SwiftUI based interface

Data in interfaces

# The iOS UI: SwiftUI

Our iOS UI uses SwiftUI

We create structures which inherit from View

Then arrange them as a tree

- Horizontal layouts use an HStack
- Vertical layouts use a VStack

The Kingfisher library is used to load images

- The type is KFImage

```
✓ iosApp
  > Configuration
  ✓ iosApp
    > Assets.xcassets
    > Preview Content / Preview Assets.xcassets
    ✎ ContentView.swift
    ✎ CountriesView.swift
    ✎ Country.swift
    ✎ CountryDetailsView.swift
    ✎ CountryRowView.swift
    </> Info.plist
    ✎ iOSApp.swift
    ✎ WeatherView.swift
  > iosApp.xcodeproj
```

# The CountryDetailsView

VStack with two Text views



```
struct CountryDetailsView: View {
    @State var country: Country

    var body: some View {
        HStack(alignment: .center, spacing: 0) {
            KFImage
                .url(URL(string: country.flags.png))
                .setProcessor(DownsamplingImageProcessor(size: CGSizeMake(75.0, 75.0)))
                .frame(width: 75, alignment: .top)
                .border(Color.gray)
                .padding(15)
            VStack(alignment: .leading) {
                Text(country.name.common).font(.body).fontWeight(.bold)
                Text(country.name.official).font(.caption)
            }.frame(alignment: .bottom)
        }.frame(maxWidth: .infinity, alignment: .leading)
    }
}
```

# Data In Interfaces: Android

```
var listCountries: List<Country> by remember { mutableStateOf(mutableListOf()) }

LaunchedEffect(Unit) { this: CoroutineScope
    listCountries = CountrySDK().getCountries()
}

Column { this: ColumnScope
    LazyColumn() { this: LazyListScope
        itemsIndexed(items = listCountries) { this: LazyItemScope index: Int, item: Country →
            CountryCard(
                modifier = Modifier,
                country = item,
                currentCountry = index == 0
            )
        }
    }
}
```

# Data In Interfaces: iOS

```
// In shared/build.gradle.kts

plugins {
    id("com.google.devtools.ksp") version "1.9.20-1.0.14"
    id("com.rickclephas.kmp.nativecoroutines") version "1.0.0-ALPHA-20"
}

sourceSets {
    all {
        languageSettings.optIn("kotlin.experimental.ExperimentalObjCName")
    }
}
```

# Data in interfaces: iOS

```
// In shared/commonMain/kotlin/network/CountryApi.kt

@NativeCoroutines
suspend fun getAllCountries(): List<Country> {
    return httpClient.get("https://restcountries.com/v3.1/all")
        .body<List<Country>>()
        .sortedBy { it.name.common }
}
```

# Data in interfaces: iOS

```
// In iosApp/iosApp/CountriesView.swift

func loadCountries() {
    Task {
        do {
            self.loadableCountries = .loading
            let countries = try await asyncFunction(for: api.getAllCountries())
            self.loadableCountries = .result(countries)
        } catch {
            self.loadableCountries = .error(error.localizedDescription)
        }
    }
}
```

State displayed on UI

# What do we need?

Domain types ✓

Networking code ✓

Support for caching ✓

Platform specific support:

- For creating the cache file ✓
- For working with locations ✓

A JetPack Compose based interface ✓

A SwiftUI based interface ✓

Data in interfaces ✓

**Kotlin-Swift interopedia**

## Introduction

Kotlin/Native provides bidirectional interoperability with Objective-C. At the time of writing, Kotlin is not directly interoperable with Swift but rather indirectly via an Objective-C bridge. Swift export is something that the Kotlin/Native team intends to address in future. However, the reason for choosing to start with Objective-C is sound: older projects containing Objective-C code can also call shared Kotlin code, along with projects containing Swift code.

In general, the basics of the Kotlin language such as classes, properties, and functions can be easily used from Swift. However, some other language features may not be as readily used. This interoperability encyclopedia (or "interopedia") aims to explain how shared Kotlin code using various language features can be called from Swift. When there is no straightforward way to do so, we discuss workarounds and library solutions, if available.

In order to provide the best possible experience for Swift developers, the rule of thumb is that using the simplest language features is best. Kotlin developers may not be expert Swift developers as well, so collaboration with their Swift teammates is required so that Kotlin developers can create shared Kotlin APIs that can be called in a beautifully idiomatic way.

## How to use

## Interopedia

Packages

No packages published

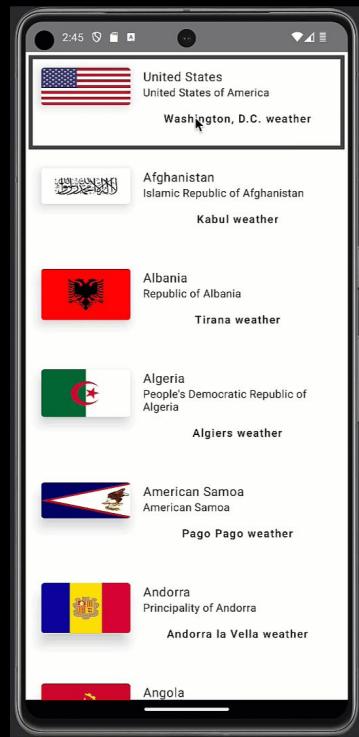
Languages

Swift 66.8%    Kotlin 33.2%

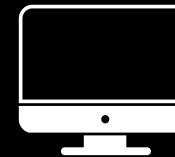
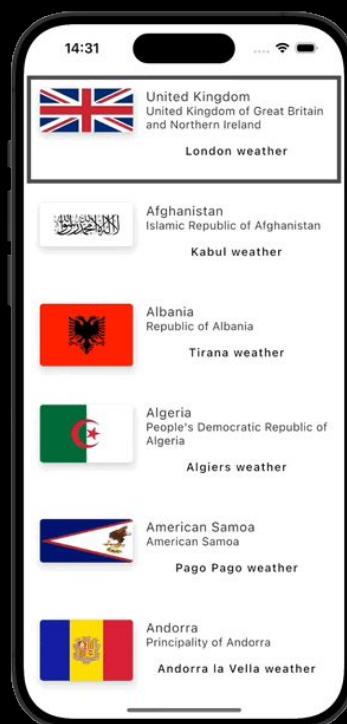
# Building an App (with Shared UI)



Android + CM



iOS + CM



Desktop + CM



Android

iOS

Desktop

## Compose App Module

UI

(Views)

*Presentation*

Business / Domain

(Entities, Use Cases, Interactors)

Data / Core

(Repositories, HTTP Clients, Cache)

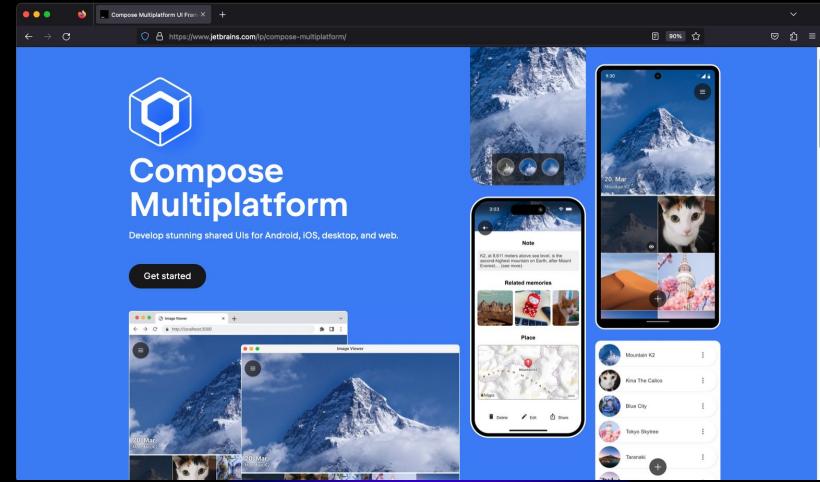
Multiplatform  
Libraries

This will be eas(y|ier)

# Moving to a shared UI

Our UI will now be implemented as common code

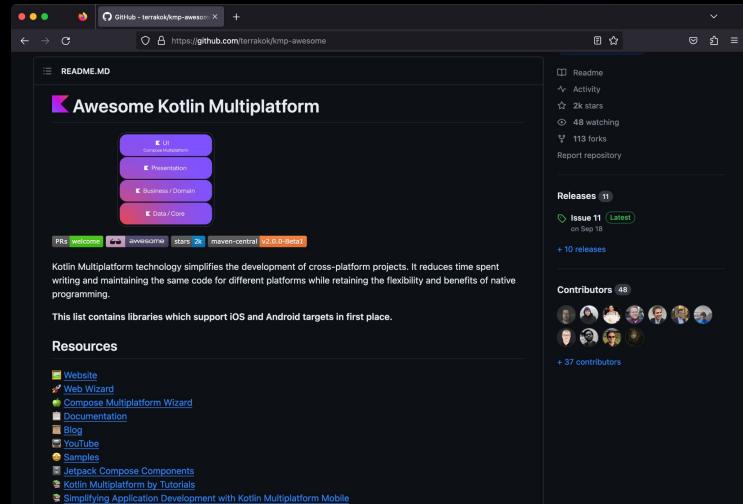
- The SwiftUI version is no longer required
- The Jetpack Compose version is ported



# Moving to a shared UI

To move from Jetpack Compose to Compose Multiplatform

- We remove platform specific dependencies
- Multiplatform Libraries take their place
- We ensure all features are supported



 **Kotlin Multiplatform Wizard**

New Project    Template Gallery (Coming soon)     

Project Name: WebinarProject

Project ID: com.kmp.webinar

**Android**   
With Compose Multiplatform UI framework based on Jetpack Compose

**iOS**   
UI Implementation  
 Share UI (with Compose Multiplatform UI framework) (Alpha)  
 Do not share UI (use only SwiftUI)

**Desktop**

**Web** (Coming soon)   
Kotlin/Wasm is maturing rapidly. Soon, we'll enable Web project creation with Compose for Web in the wizard. In the meantime, check out these examples on GitHub for how to use Kotlin/Wasm.

**Server**

**DOWNLOAD**



**Android**   
With Compose Multiplatform UI framework based on Jetpack Compose

**iOS**   
UI Implementation  
 Share UI (with Compose Multiplatform UI framework) (Alpha)  
 Do not share UI (use only SwiftUI)

**Desktop**   
With Compose Multiplatform UI framework

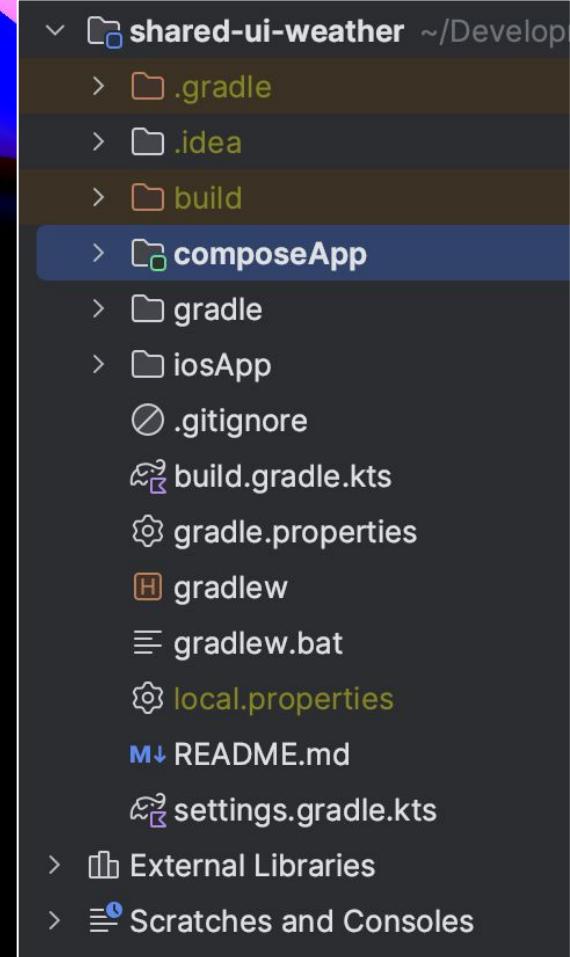
# The project structure

We no longer have a **shared** module

The **composeApp** module is all we need

An iOS project still contains launcher code

- But we no longer use it directly

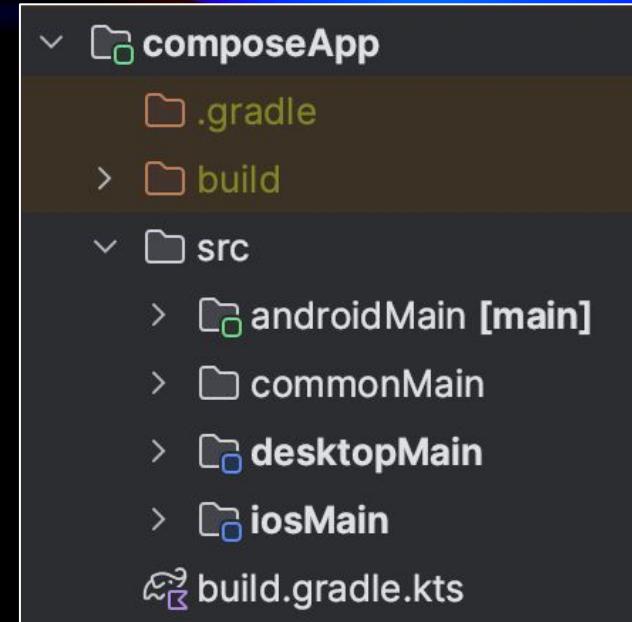


# The composeApp module

The module contains four source sets

A **commonMain** source set for shared code

One source set for each targeted platform



# The commonMain source set

The shared UI lives inside `commonMain`

We create a new package for our composables

```
└ commonMain
  └ kotlin [commonMain] sources root
    > cache
    > country
    > location
    > network
    < ui
      └ App.kt
      └ Country.kt
      └ CountryCard.kt
      └ CountryNames.kt
      └ Flag.kt
      └ HomeScreen
      └ WeatherButton.kt
      └ WeatherCard.kt
      └ WeatherScreen
    > weather
```

# Changing Dependencies

We need to replace platform specific libraries in our UI

In this case we:

- Replace Coil with Compose ImageLoader
- Replace Navigation Compose with Voyager

# Changing dependencies

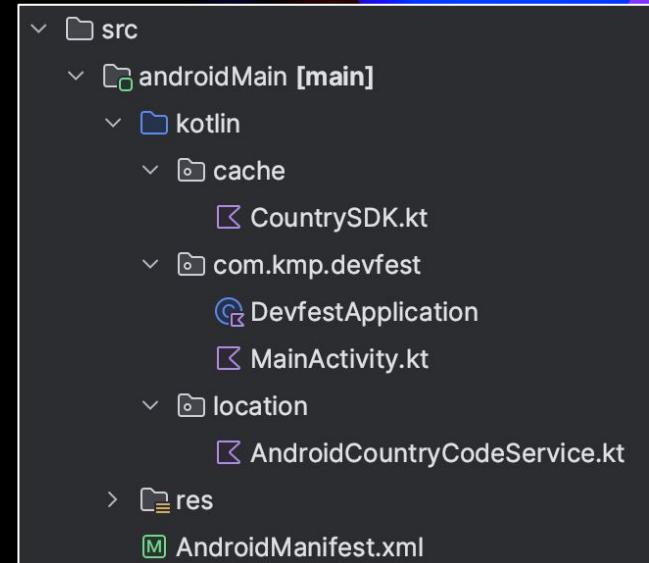
```
androidMain.dependencies {  
    ...  
  
    implementation("io.coil-kt:coil-compose:2.5.0")  
    implementation("androidx.navigation:navigation-compose:2.7.5")  
}  
  
commonMain.dependencies {  
    ...  
  
    api("io.github.qdsfdhv:hive-image-loader:1.6.3")  
    implementation("cafe.adriel.voyager:voyager-navigator:1.0.0-rc05")  
}
```

# The androidMain source set

This source set is now much smaller

We only require:

- An activity, to launch the root composable
- An application, to set the path for caching
- Actual declarations for expected functions



# The androidMain source set

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        setContent {  
            App()  
        }  
    }  
}
```

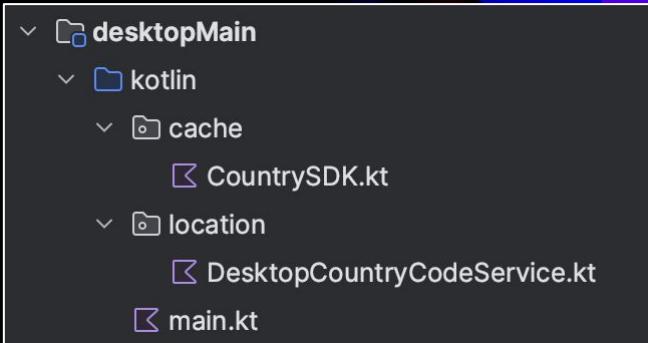
# The androidMain source set

```
class DevfestApplication : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
  
        filePath = "${filesDir.path}/country_cache.json"  
    }  
}
```

# The desktopMain source set

For the desktop we require:

- Actual declarations as before
- A main method, which launches a window



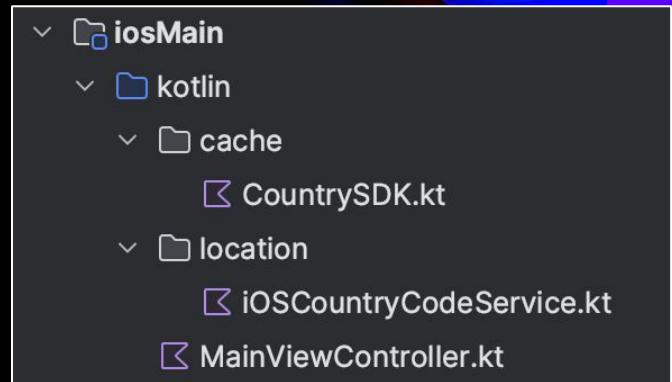
# The desktopMain source set

```
val state = WindowState(  
    size = DpSize(450.dp, 700.dp),  
    position = WindowPosition(200.dp, 100.dp)  
)  
  
fun main() = application {  
    Window(  
        title = "Devfest Weather (Desktop)",  
        onCloseRequest = ::exitApplication,  
        state = state  
    ) {  
        App()  
    }  
}
```

# The iosMain source set

For iOS we require:

- Actual declarations as before
- A view controller, to be invoked from Swift



# The iosMain source set

```
fun MainViewController() = ComposeUIViewController { App() }
```

```
//in iosApp/iosApp/ContentView.swift
struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) → UIViewController {
        MainViewControllerKt.MainViewController()
    }

    func updateUIViewController(_ uiViewController: UIViewController, context: Context) {}
}
```

# Conclusions

# Conclusions

Both the Native UI and Shared UI stories work really well

- You can fine tune which code you want to share

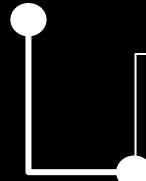
You have access to an ecosystem of multiplatform libraries

- But once again, you only use what you want

# Conclusions

There is a lot more still to come!

- Fleet will provide a polyglot IDE, ideal for KMP
- Compose Multiplatform will improve and expand
- Amper will simplify project configuration

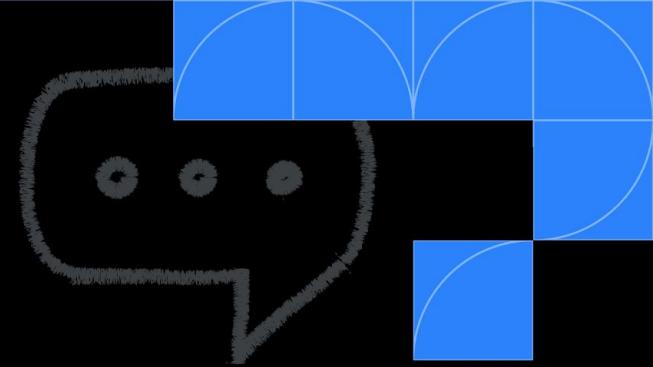


```
# Produce a shared library for the JVM, Android, and iOS platforms:  
product:  
    type: lib  
    platforms: [jvm, android, iosArm64, iosSimulatorArm64, iosX64]
```

```
text  
'Simple Statement or URL',  
style: TextStyle(  
  color: Colors.blue[200],  
,  
)
```

# devfest

```
s.star,  
r: Colors.blue[500],  
  
Text('23'),
```



# Thank you □

garth.gilmour@jetbrains.com



@GarthGilmour