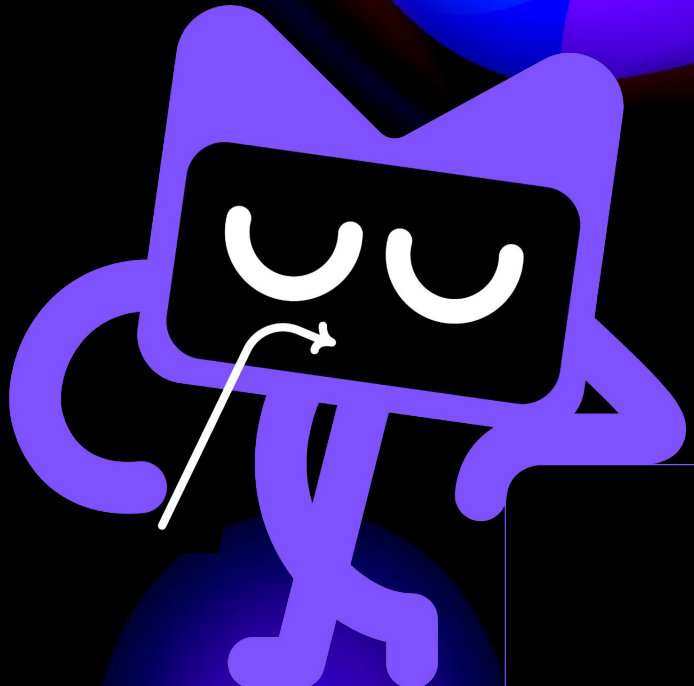
# Kotlin Multiplatform (2024 Edition)



Belfast Kotlin User Group - Workrise - 5/12/23

# Questions we will answer

What is Kotlin Multiplatform / KMP?

To get started, how do I…

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

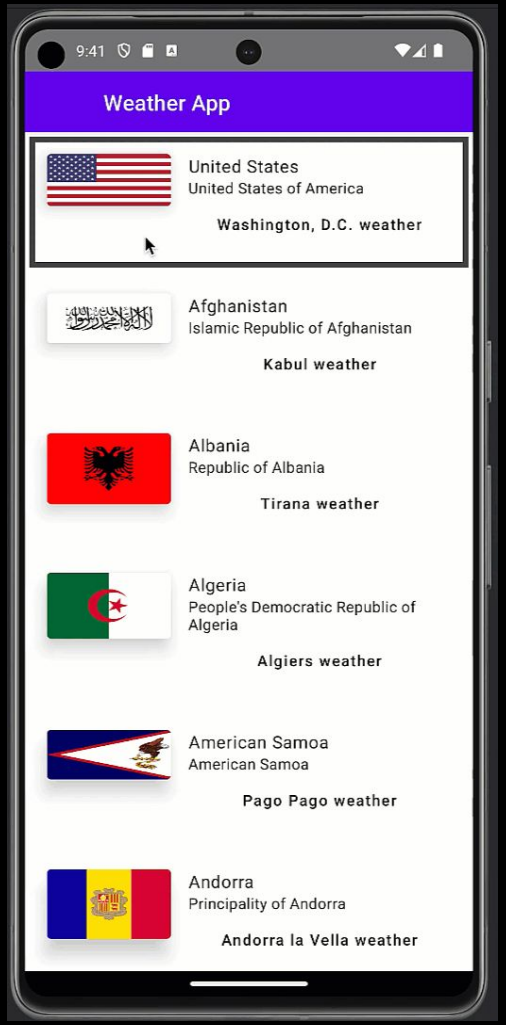
Then how do I build an app:

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

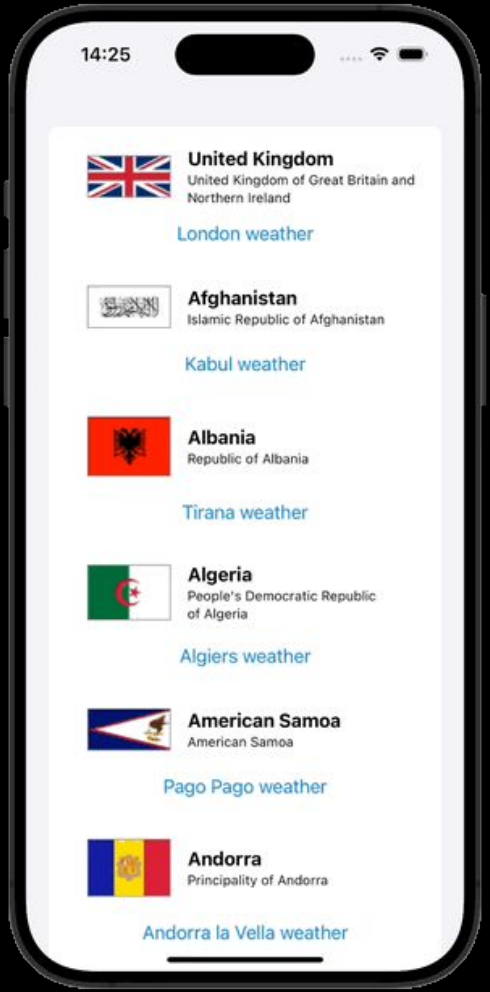
# The Weather App

## (Native UI)

Jetpack Compose



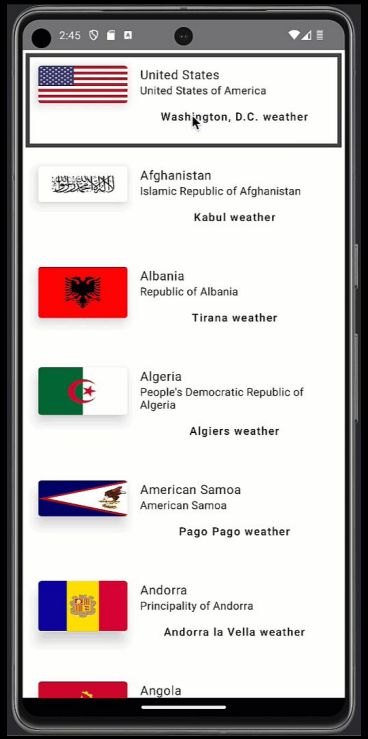

SwiftUI



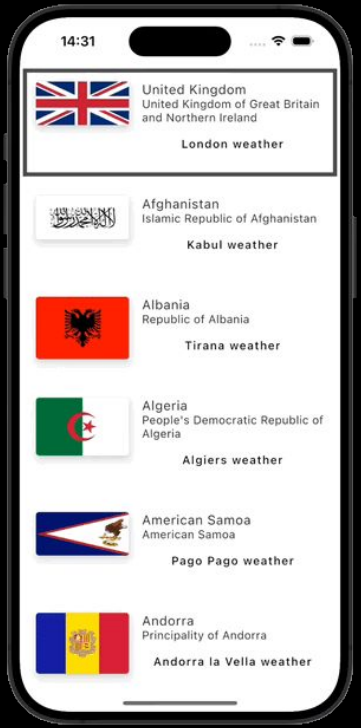
# The Weather App

## (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/belfast-kug-dec-2023

November 21, 2023
17:00 UTC

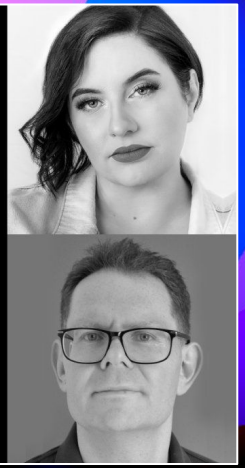
## The State of Kotlin Multiplatform

Svetlana Isakova
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 28, 2023
16:00 UTC

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

Sebastian Aigner
Márton Braun

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 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

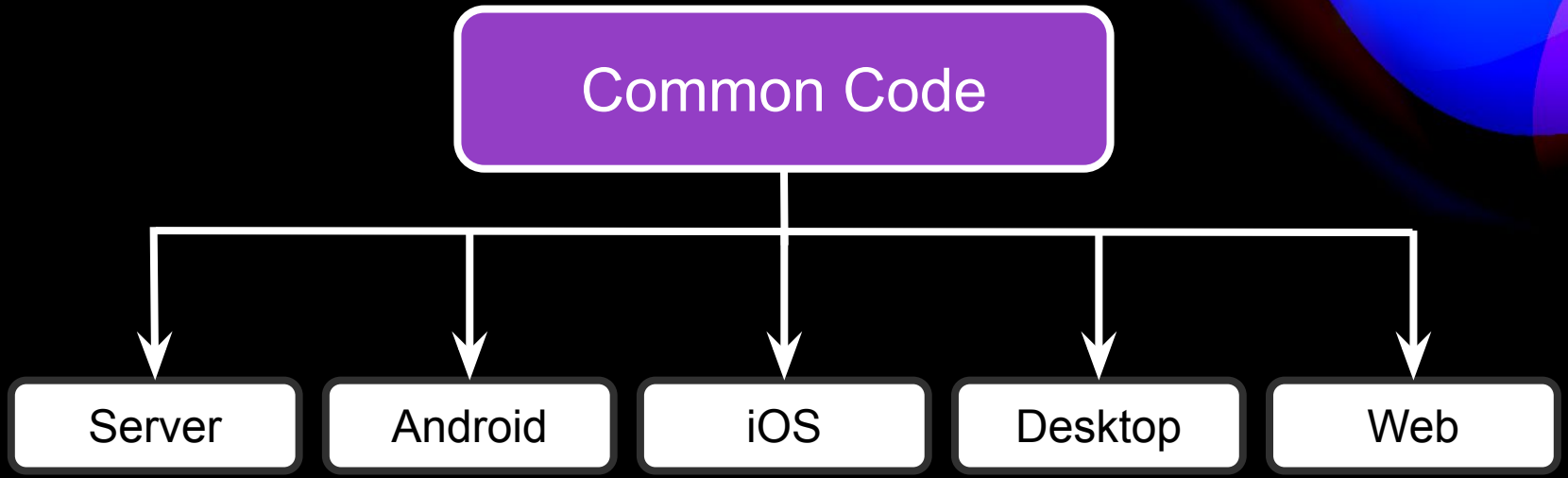
Explore our roadmap for more ⬇️ blog.jetbrains.com/kotlin/2023/11...

# What is Kotlin Multiplatform?

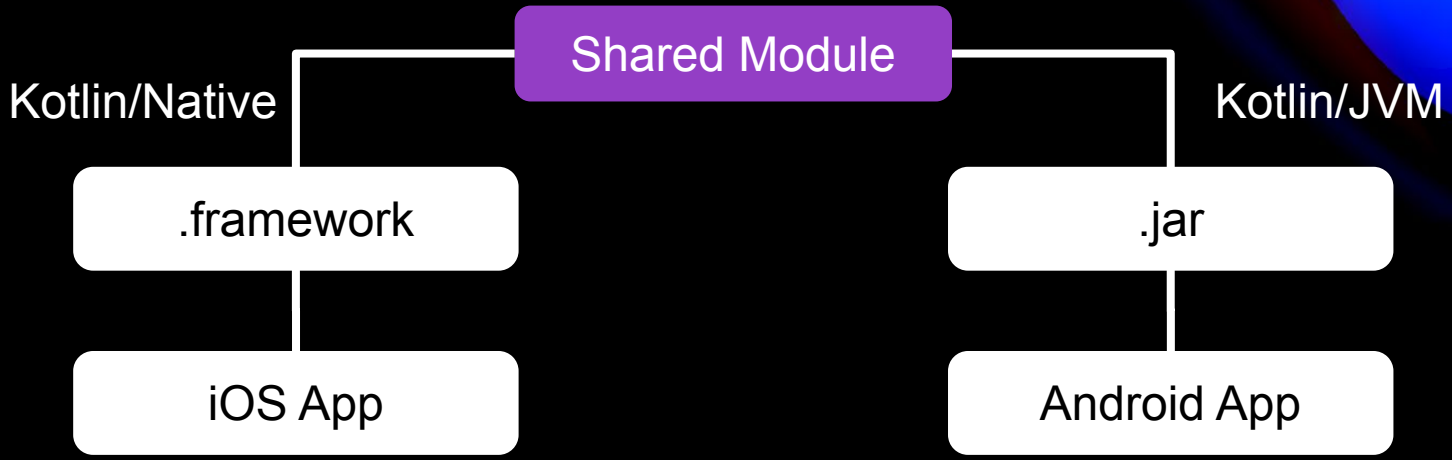
(aka. KMP)

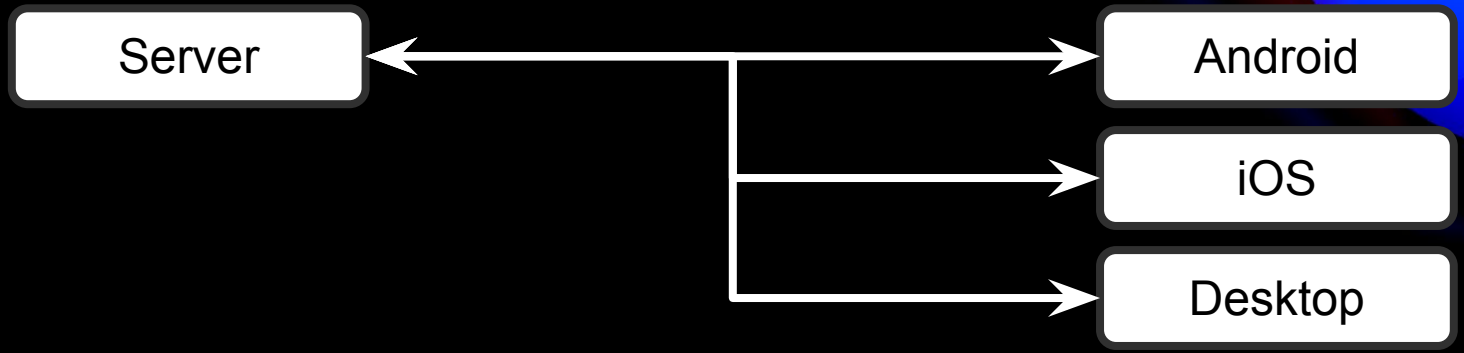
# Multiplatform Means Sharing

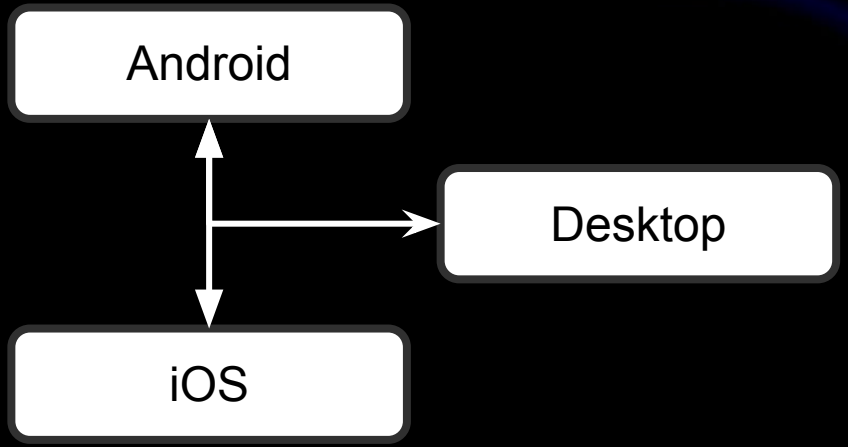
# Sharing code across platforms



# Sharing code across platforms

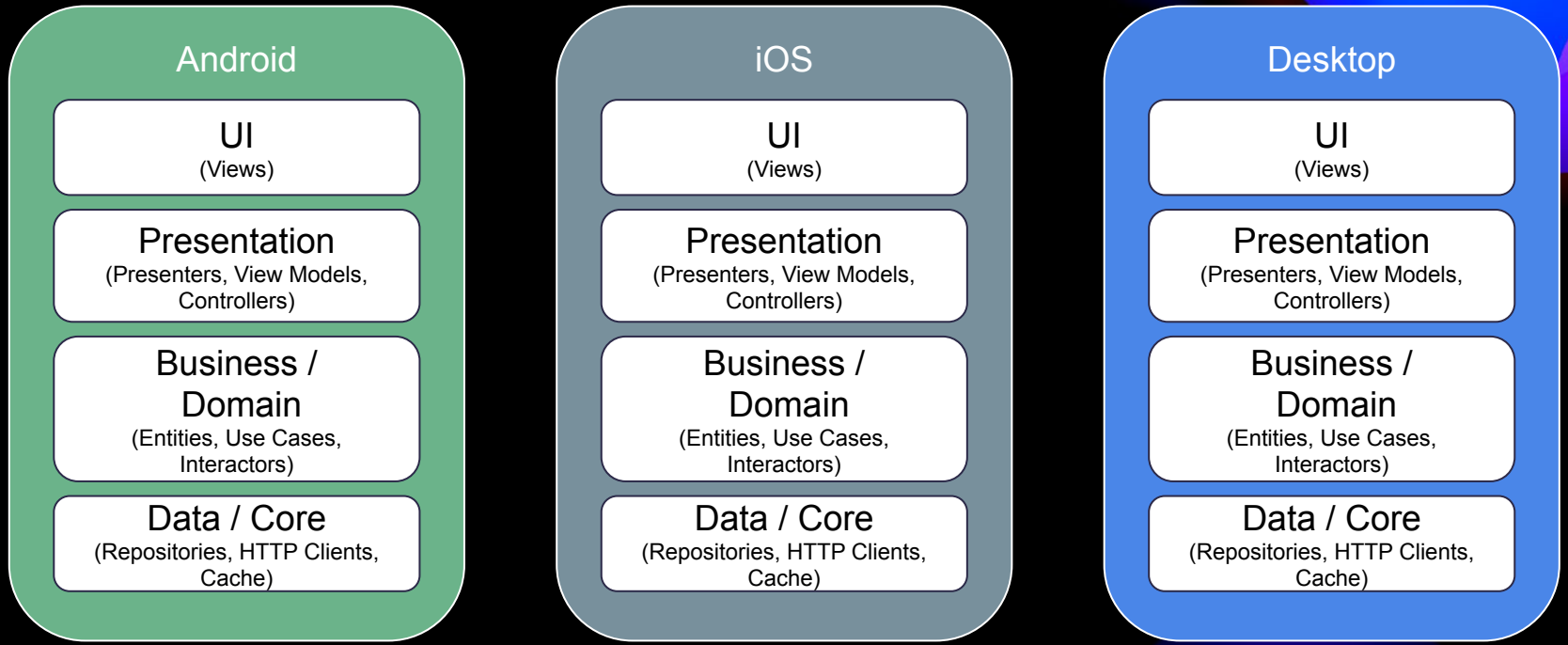



Server
Android
iOS
Desktop

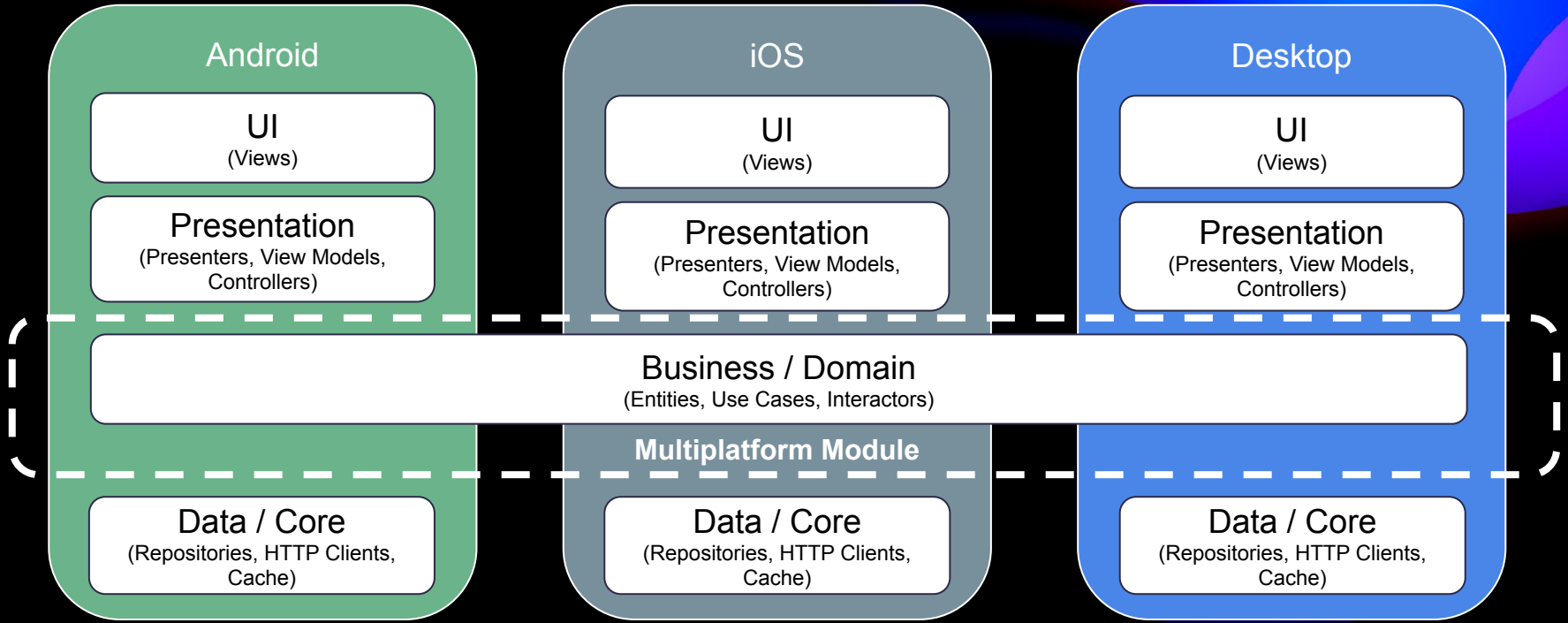

Android
Desktop
iOS

# What Do I Share?

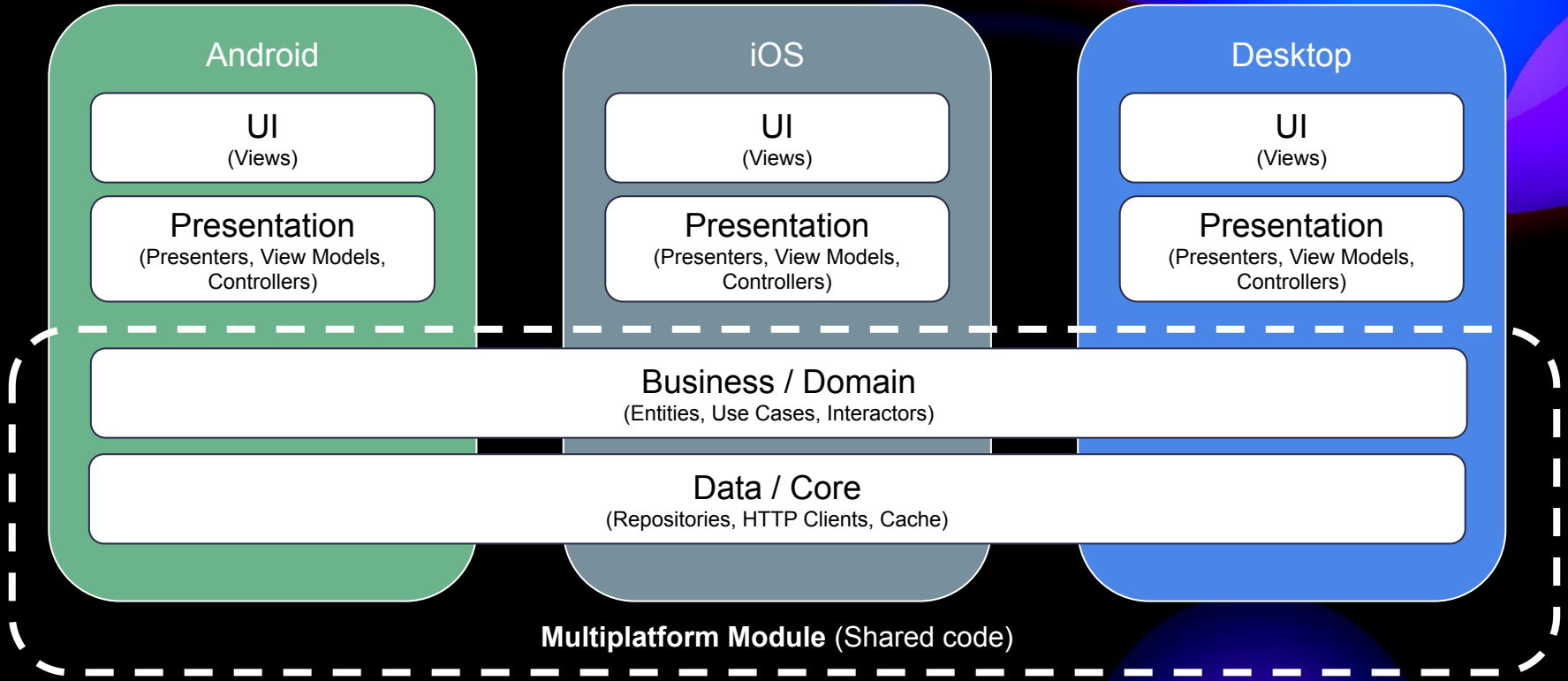
# Without KMP - separate stacks



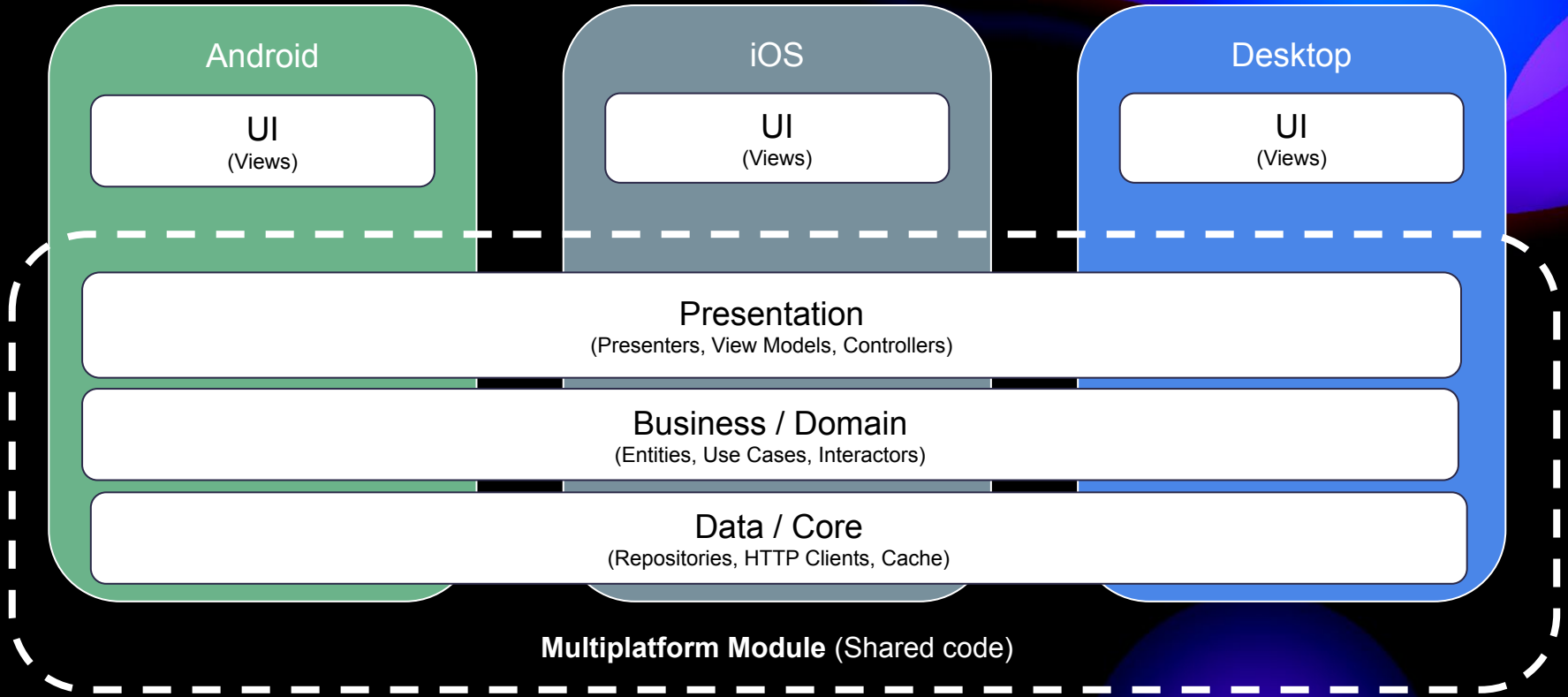
# With KMP - sharing logic & data



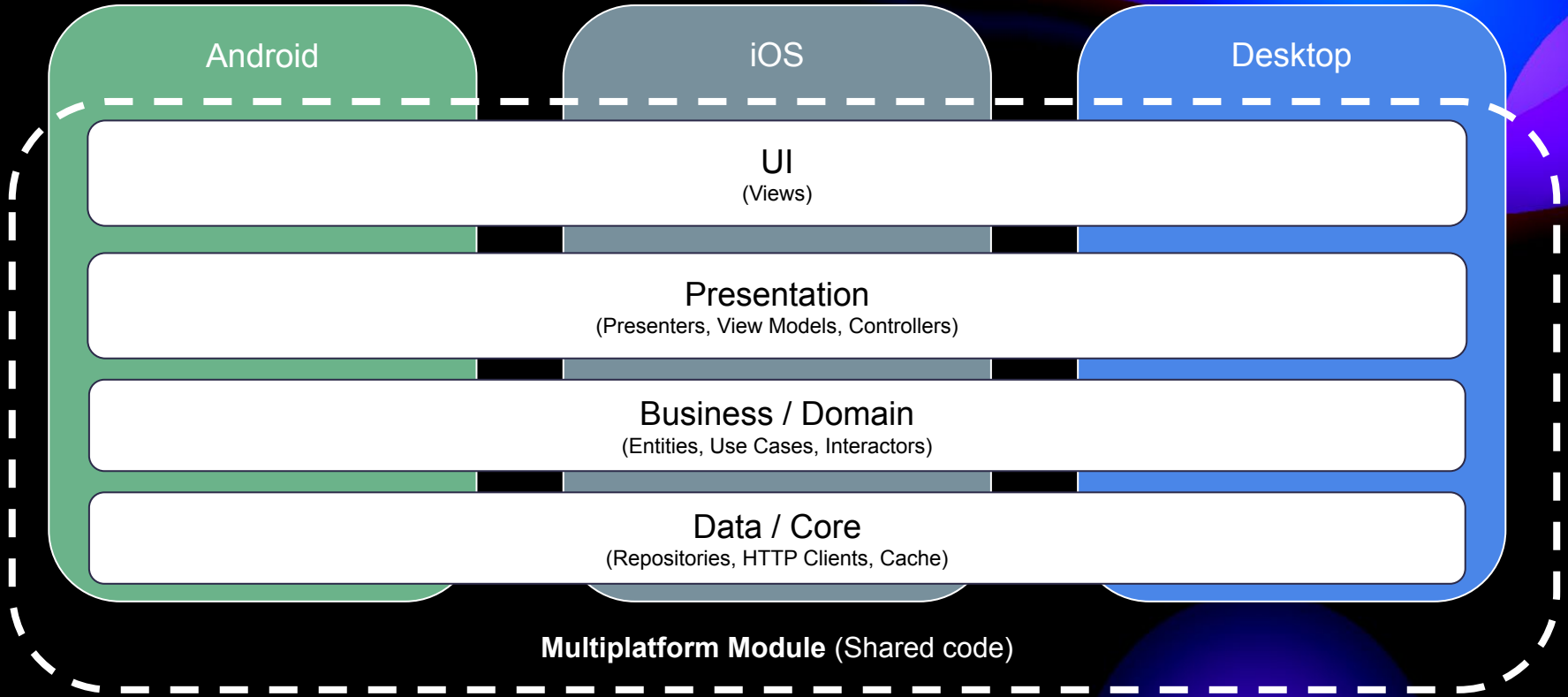
# With KMP - sharing logic, data and services



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



# With Compose Multiplatform - sharing everything



# Who Is Using KMP?


NETFLIX


PHILIPS

vmware®

AUTODESK

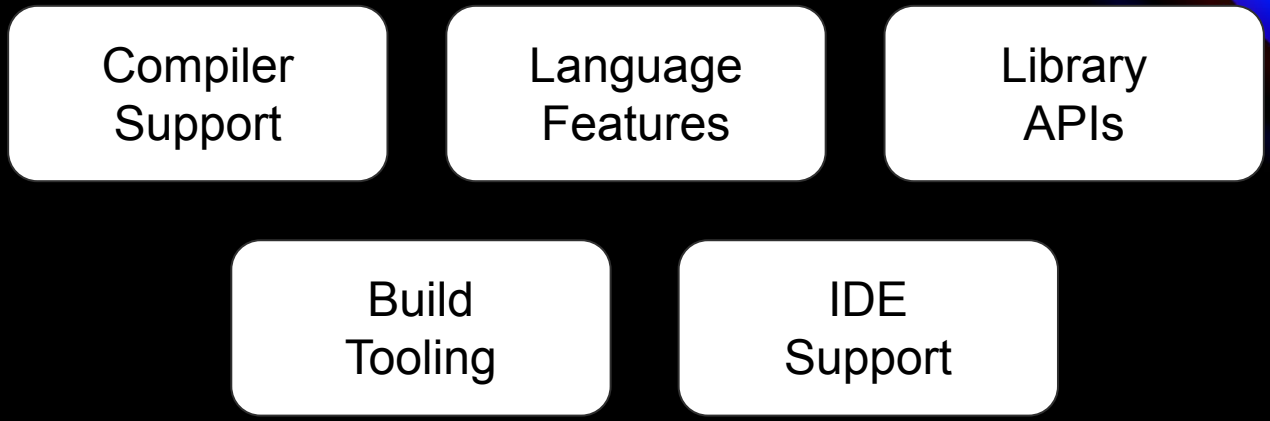

chalk

Forbes

https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html

# Define Stable?

# Strict compatibility guarantees



# Core KMP stability

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

https://kotl.in/kmp-stability

# Compose Multiplatform stability

| | |
|---|---|
| **Android** | Stable |
| **iOS** | Alpha |
| **Desktop (JVM)** | Stable |
| **Web (Kotlin/Wasm)** | Experimental |

https://kotl.in/kmp-stability

# 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.

https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-setup.html

# To target Android

Install Android Studio

Create a Virtual Device

Ignore it

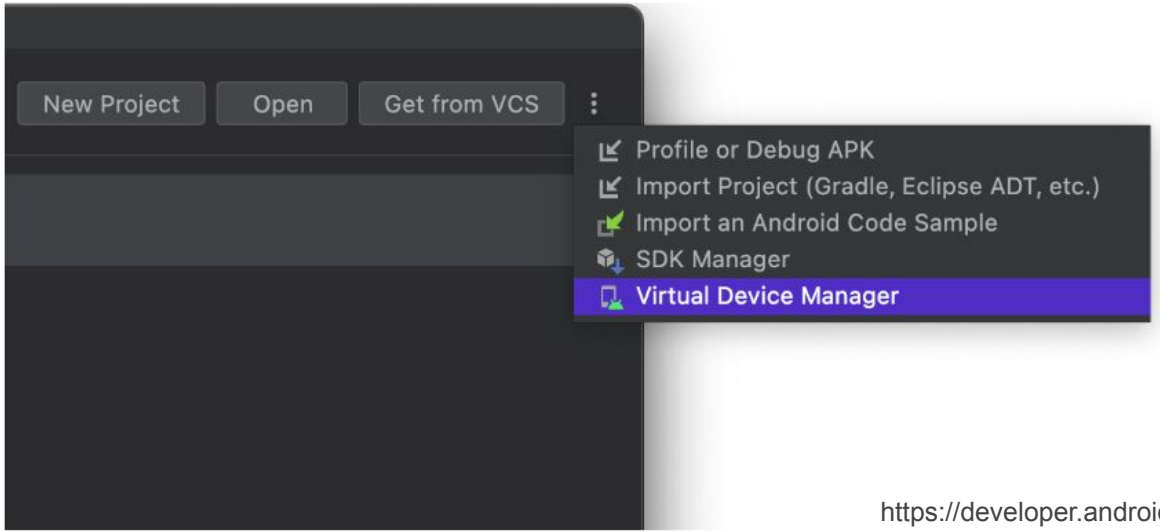
Was this helpful?

# 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**.




https://developer.android.com/studio/run/managing-avds

# 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

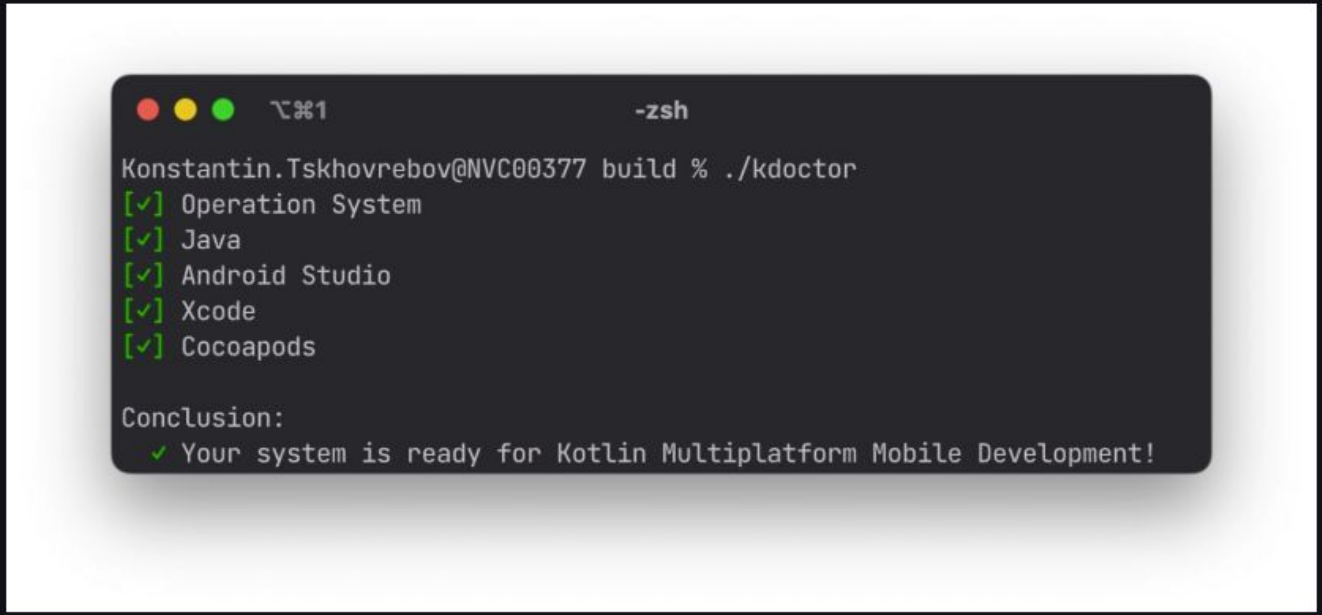
It can be used to declare native dependencies

It’s up to you is 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.



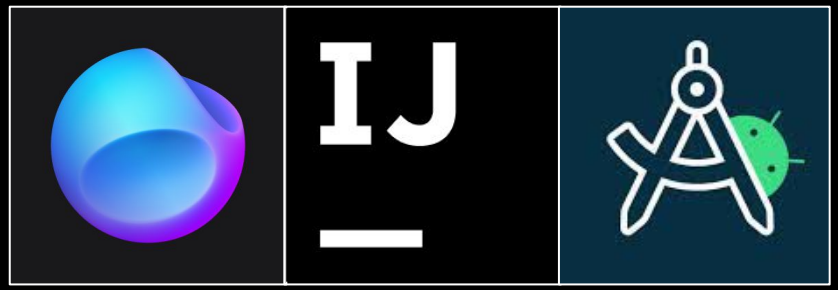

https://github.com/Kotlin/kdoctor

# 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 ←→ Kotlin)

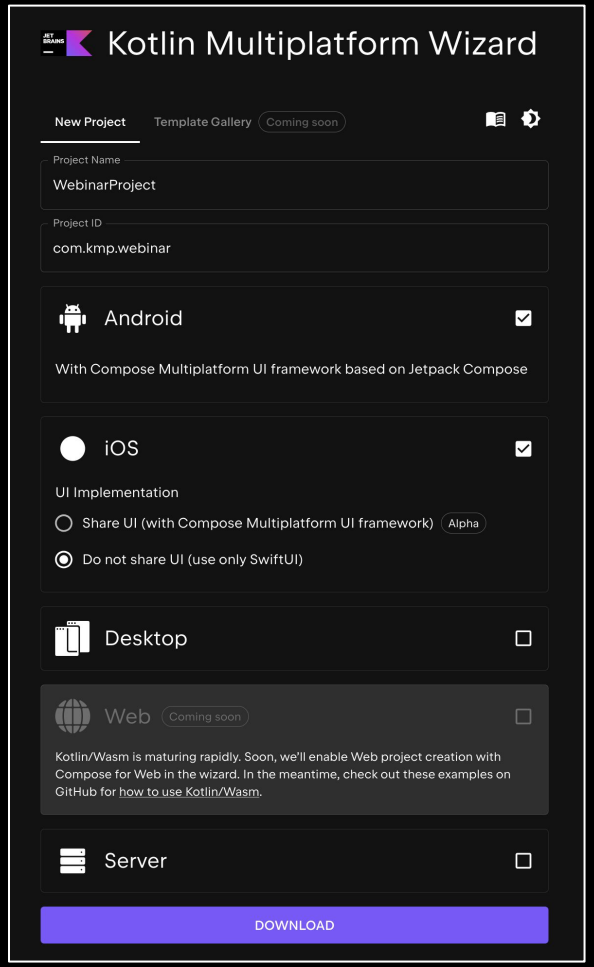
**Pro:** Cross language debugging (Swift ←→ Kotlin)

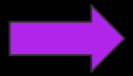
**Con:** Still in Public Preview so expect issues

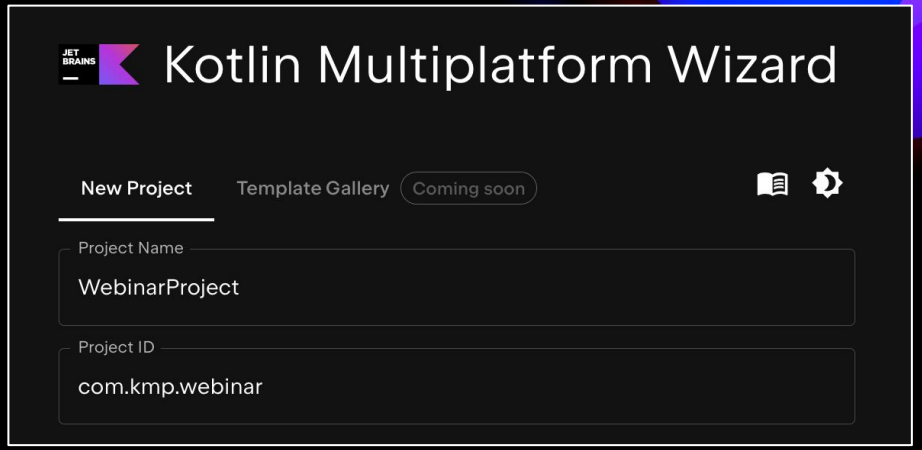
# How Do I…

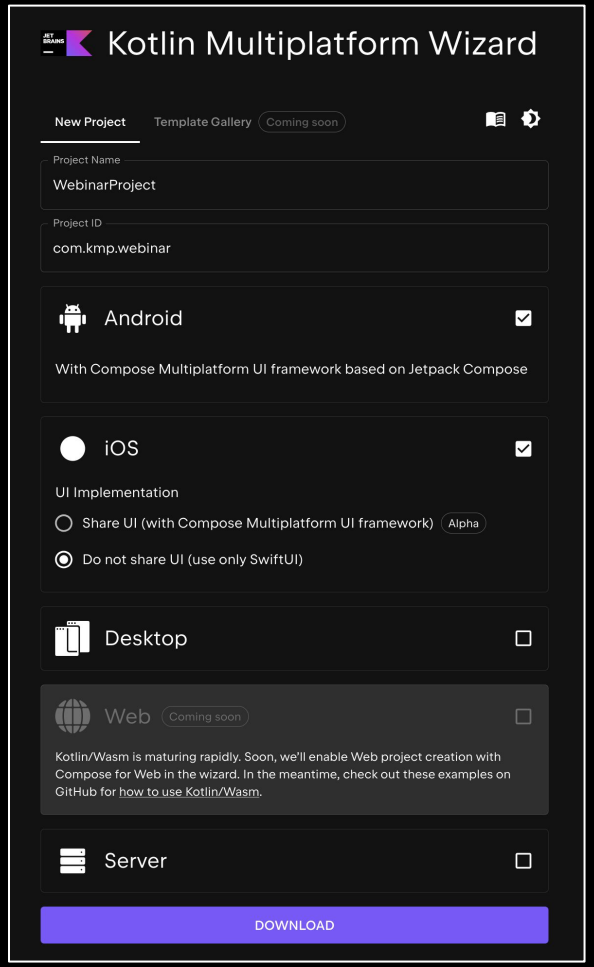
## Create a Project?

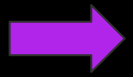
kmp.jetbrains.com

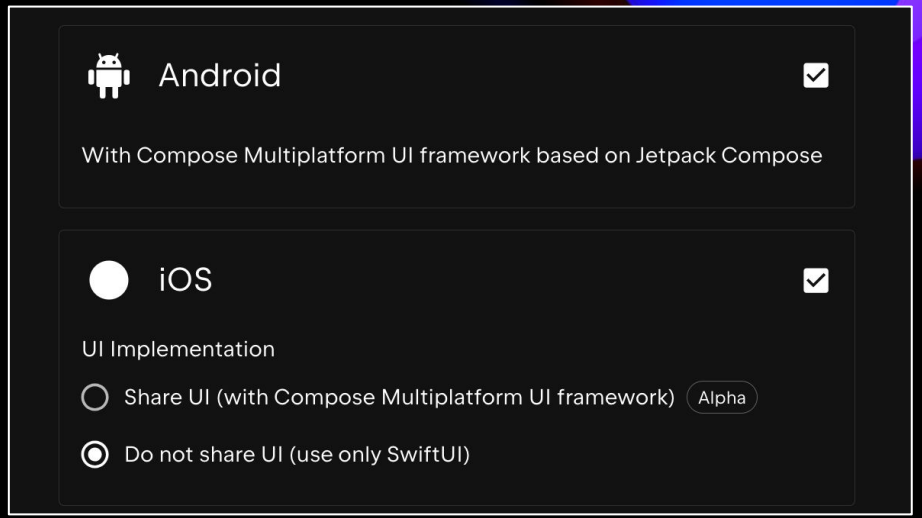

Kotlin Multiplatform Wizard
New Project
Template Gallery
Coming soon
Project Name
WebinarProject
Project ID
com.kmp.webinar
Android
With Compose Multiplatform UI framework based on Jetpack Compose
iOS
UI Implementation
Share UI (with Compose Multiplatform UI framework)
Alpha
Do not share UI (use only 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

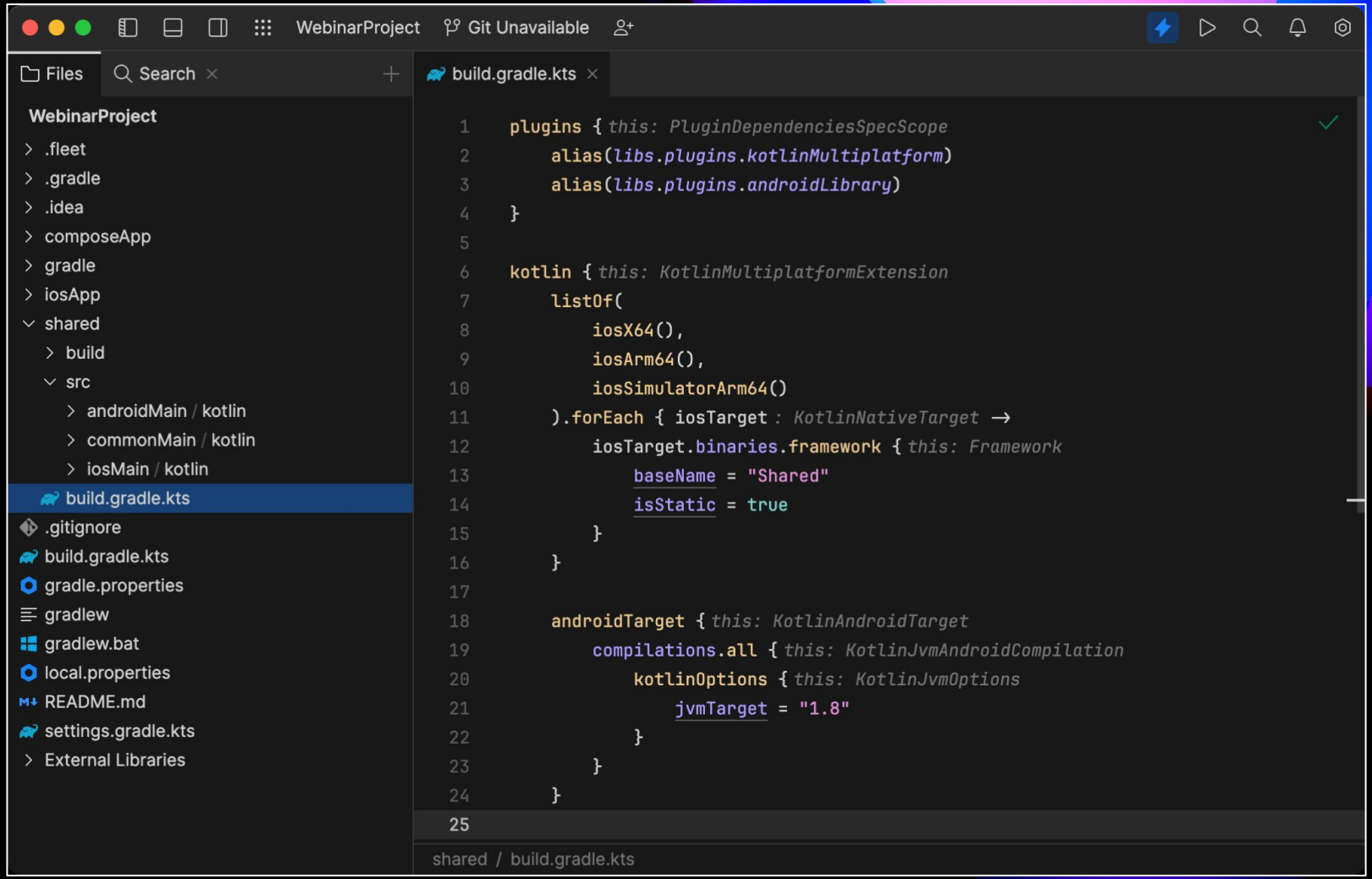



Kotlin Multiplatform Wizard
New Project
Template Gallery
Coming soon
Project Name
WebinarProject
Project ID
com.kmp.webinar


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



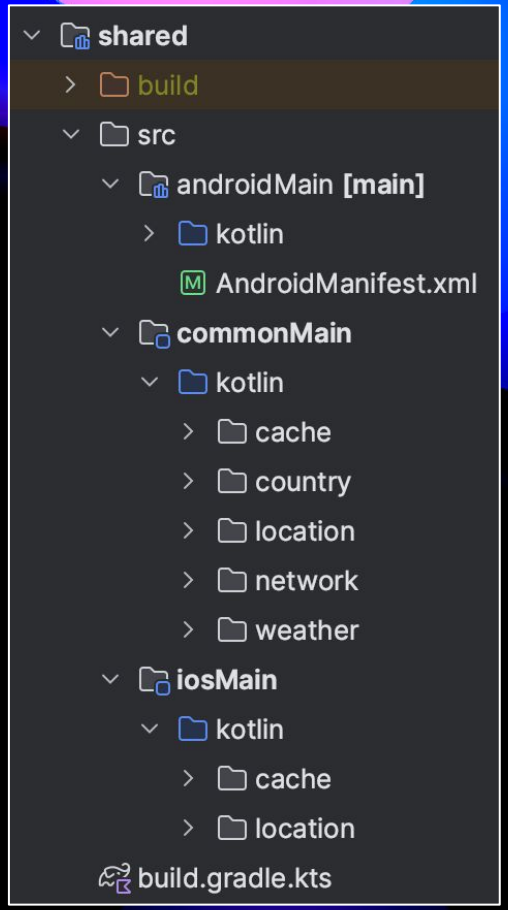

Android
With Compose Multiplatform UI framework based on Jetpack Compose
iOS
UI Implementation
Share UI (with Compose Multiplatform UI framework)
Alpha
Do not share UI (use only SwiftUI)


WebinarProject
Git Unavailable
Files
Search
build.gradle.kts
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
plugins { this: PluginDependenciesSpecScope
alias(libs.plugins.kotlinMultiplatform)
alias(libs.plugins.androidLibrary)
}
kotlin { this: KotlinMultiplatformExtension
listOf(
iosX64(),
iosArm64(),
iosSimulatorArm64()
).forEach { iosTarget : KotlinNativeTarget →
iosTarget.binaries.framework { this: Framework
baseName = "Shared"
isStatic = true
}
}
androidTarget { this: KotlinAndroidTarget
compilations.all { this: KotlinJvmAndroidCompilation
kotlinOptions { this: KotlinJvmOptions
jvmTarget = "1.8"
}
}
}
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

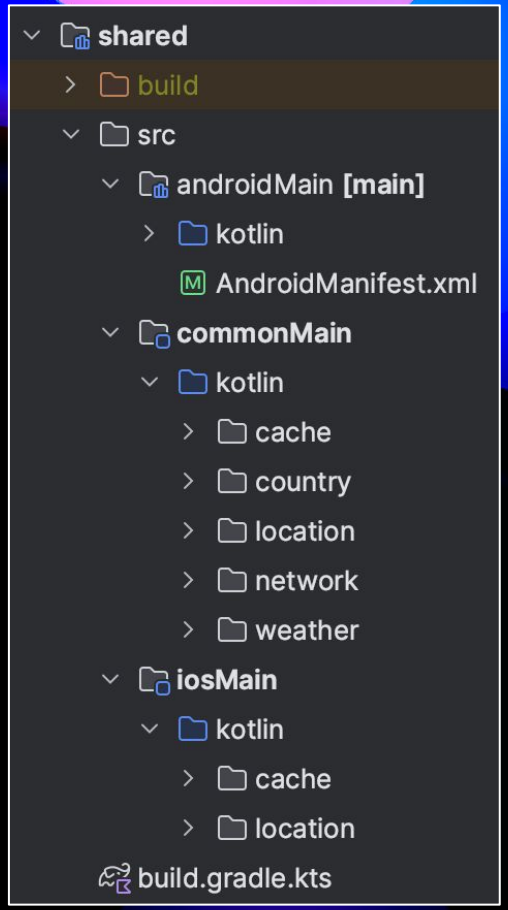


# The shared module

This is a Kotlin Multiplatform Module

It contains 3 source sets

- commonMain
- androidMain
- iosMain

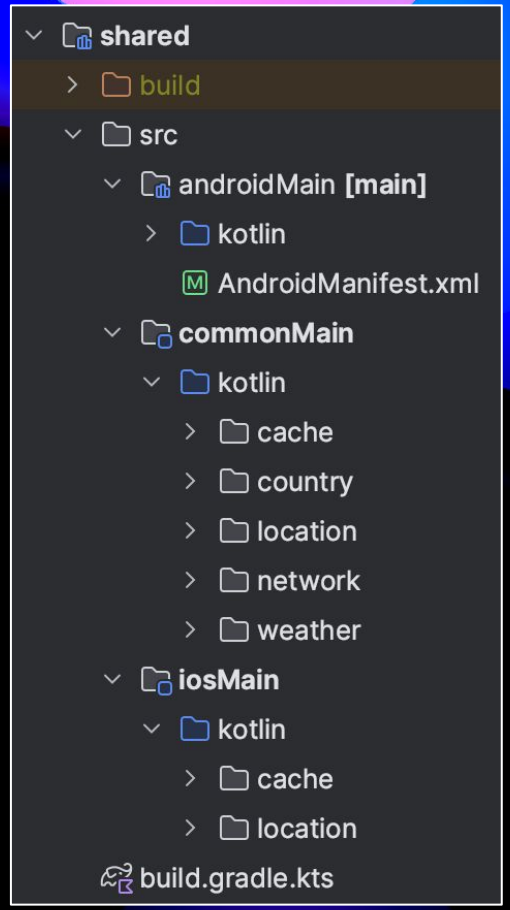


# The shared module

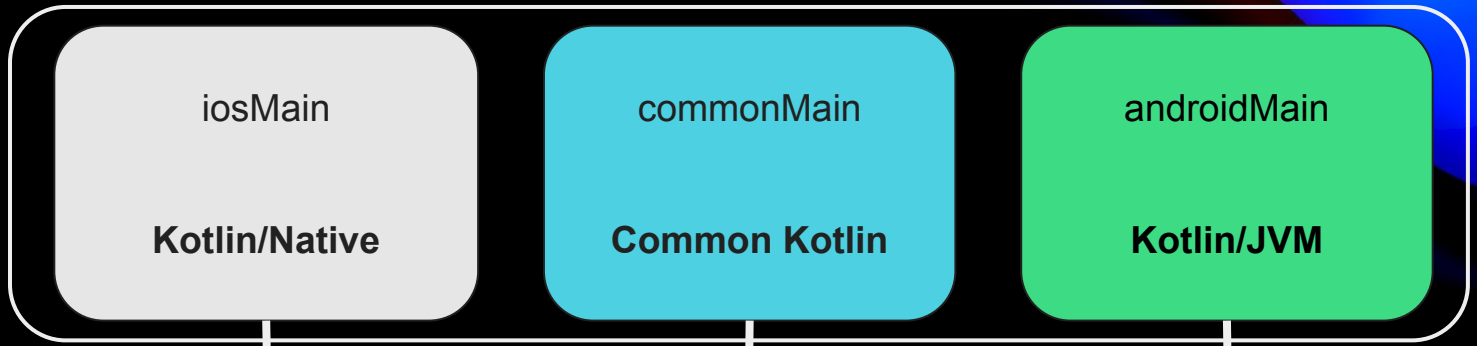
The source sets are compiled in combination:

commonMain + androidMain =

commonMain + iosMain =



# Introducing expect / actual functions




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

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

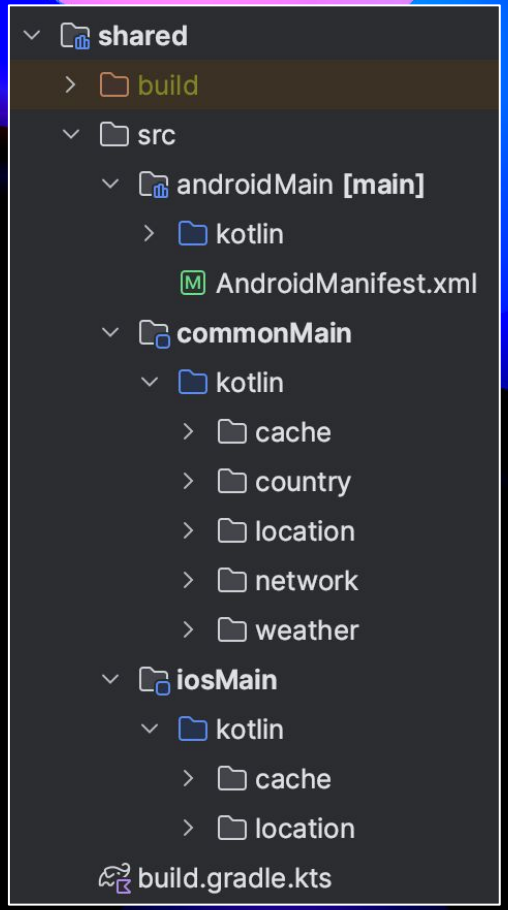
```
import java.util.*
actual fun randomUUID() =
UUID.randomUUID().toString()
```

# 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

```kotlin
interface Platform {
    val name: String
}

expect fun getPlatform(): Platform
```

```kotlin
class AndroidPlatform: Platform {
    override val name: String =
        "Android ${Build.VERSION.SDK_INT}"
}

actual fun getPlatform() = AndroidPlatform()
```

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

actual fun getPlatform() = iOSPlatform()
```

# The composeApp module

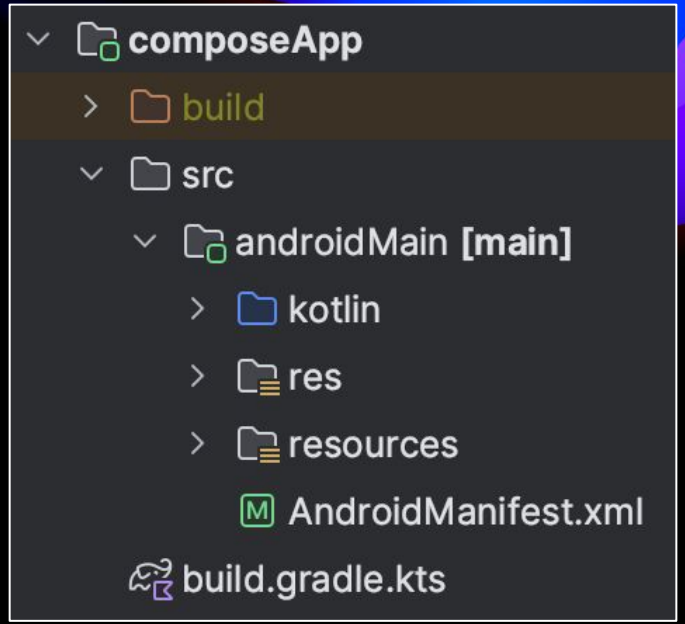
This is a Kotlin Module

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)



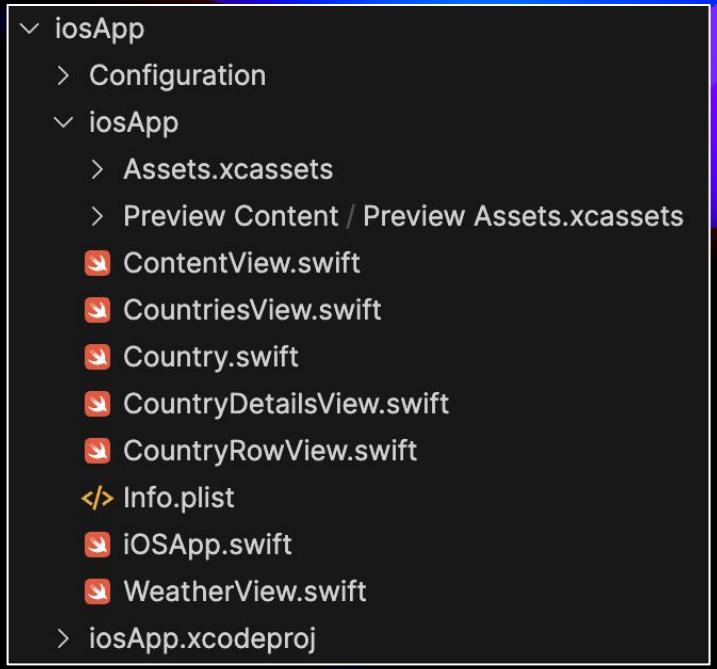
# The iosApp Folder

This is an Xcode project

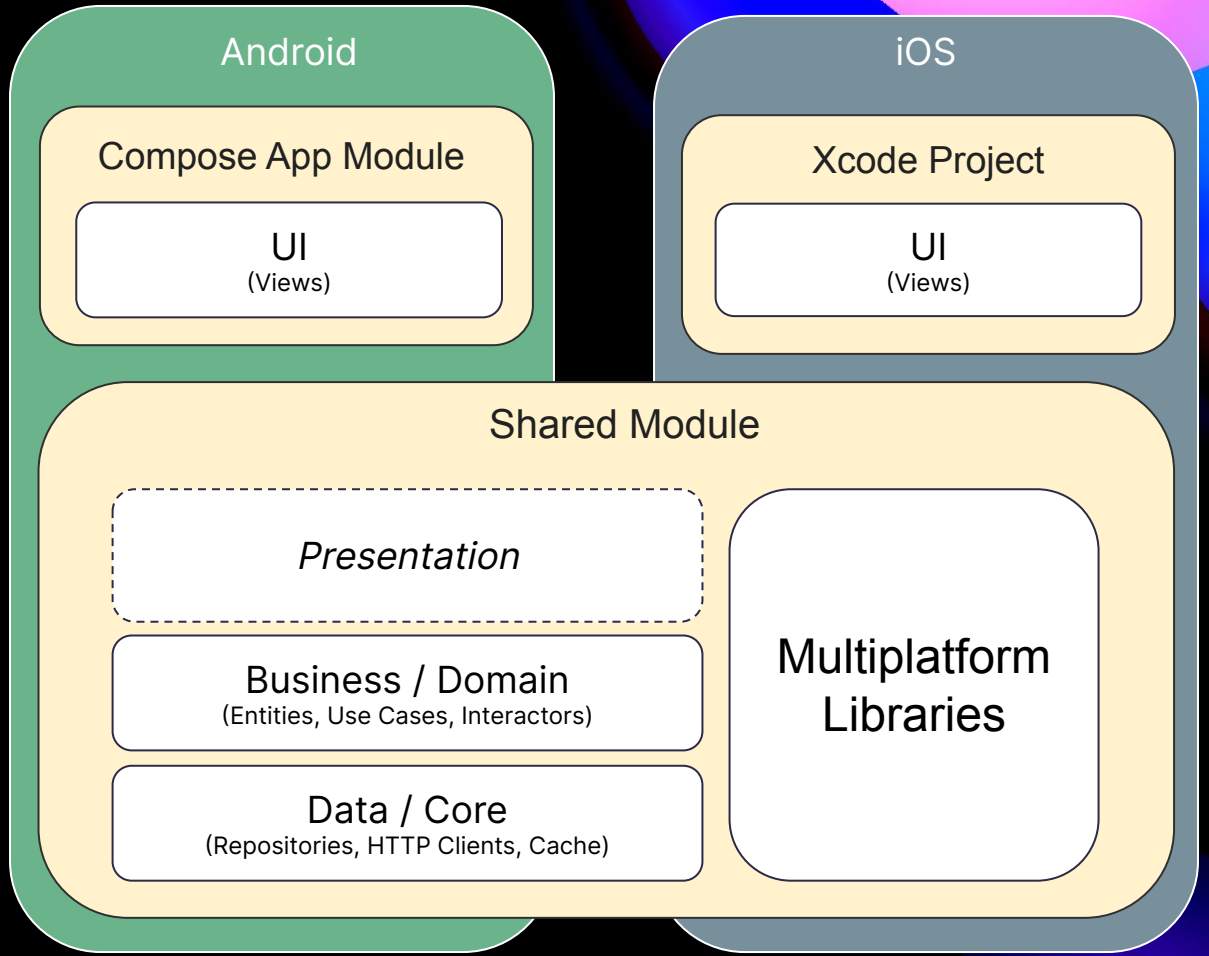
Containing the infrastructure needed to run your application on iOS

This is where we place Swift code

In this case our Native UI



# Building an App
(with Native UI)


Android
Compose App Module
UI
(Views)
iOS
Xcode Project
UI
(Views)
Shared Module
Presentation
Business / Domain
(Entities, Use Cases, Interactors)
Data / Core
(Repositories, HTTP Clients, Cache)
Multiplatform
Libraries

GitHub - terrakok/kmp-awesom

https://github.com/terrakok/kmp-awesome

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

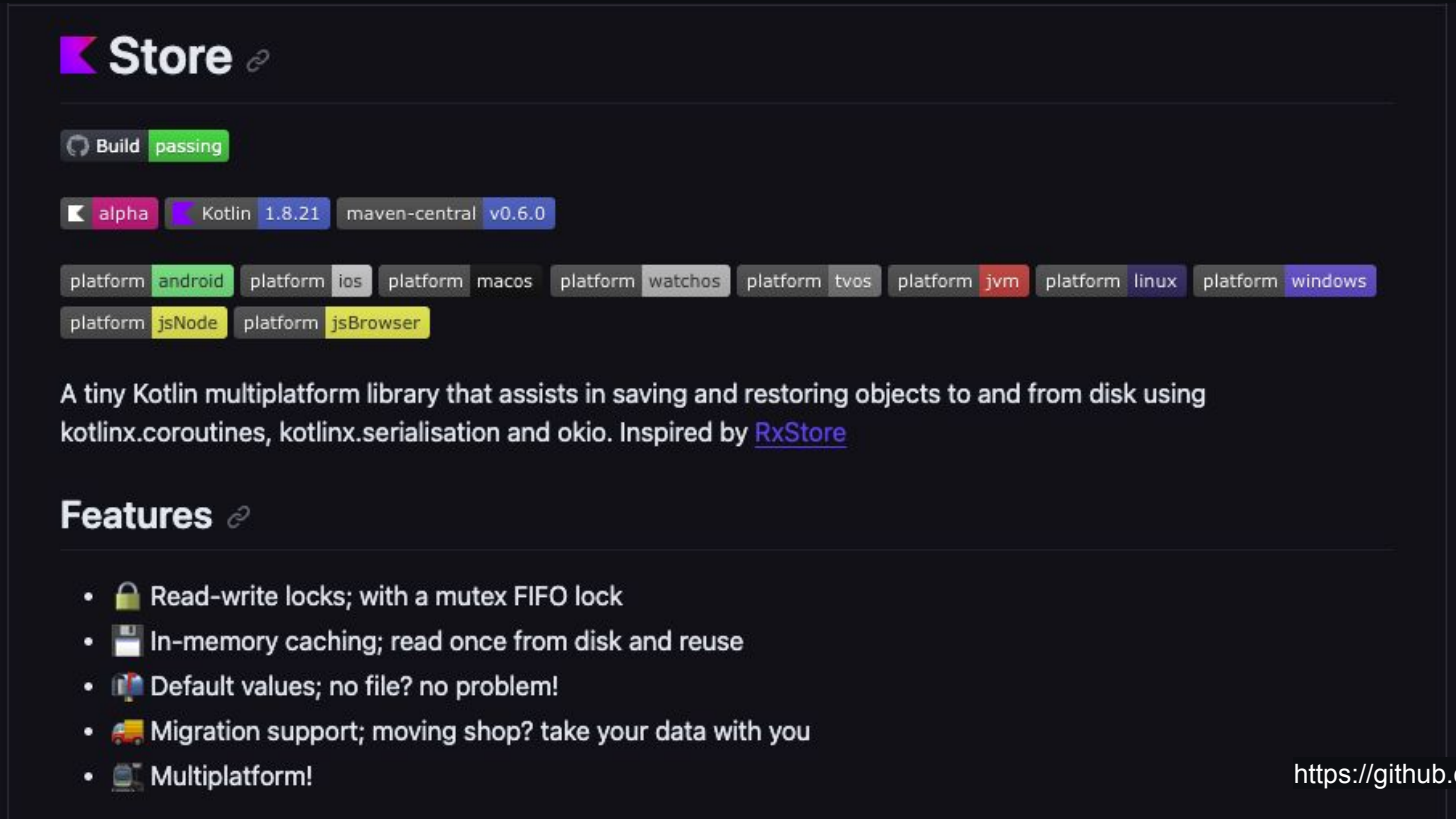
+ 10 releases

**Contributors** 48

+ 37 contributors

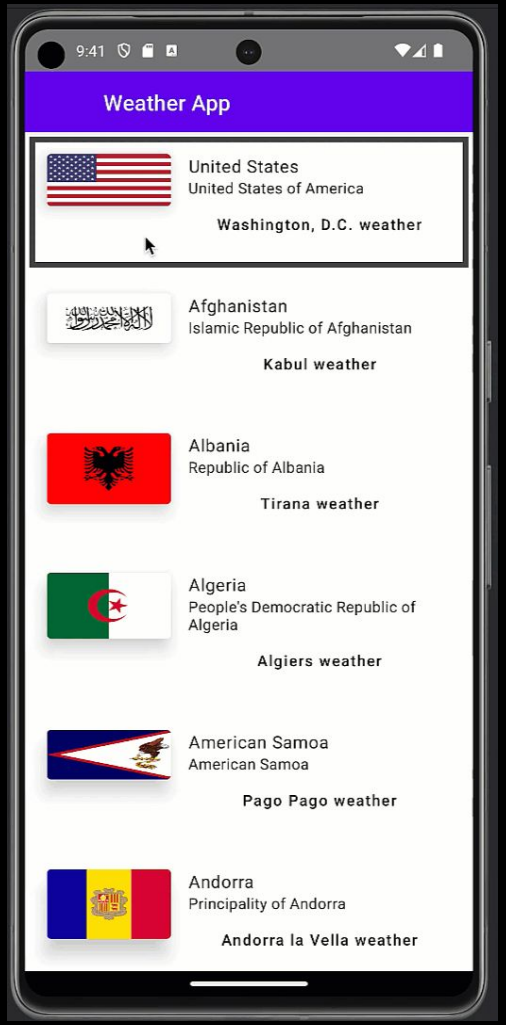
https://github.com/terrakok/kmp-awesome

# We will use KStore

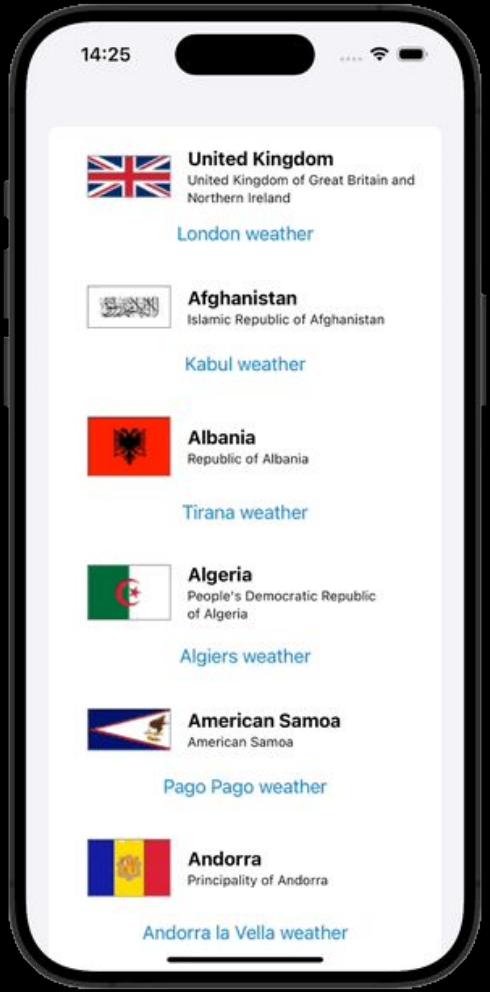



https://github.com/xxfast/KStore

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

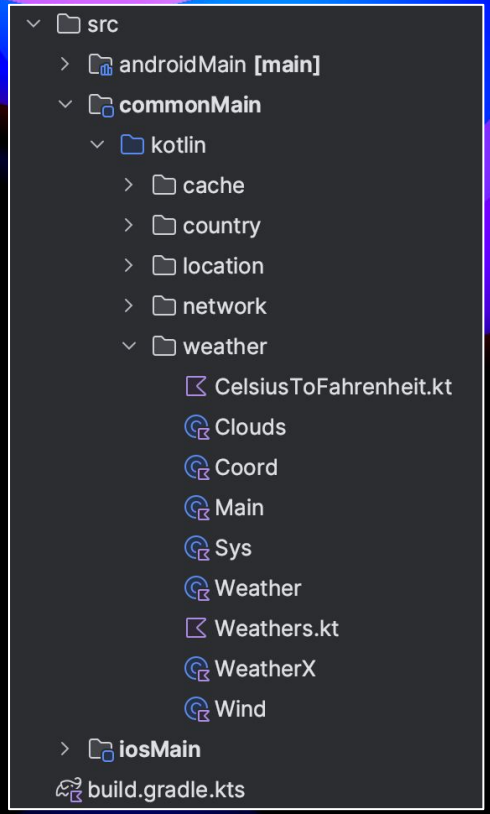
A SwiftUI based interface

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

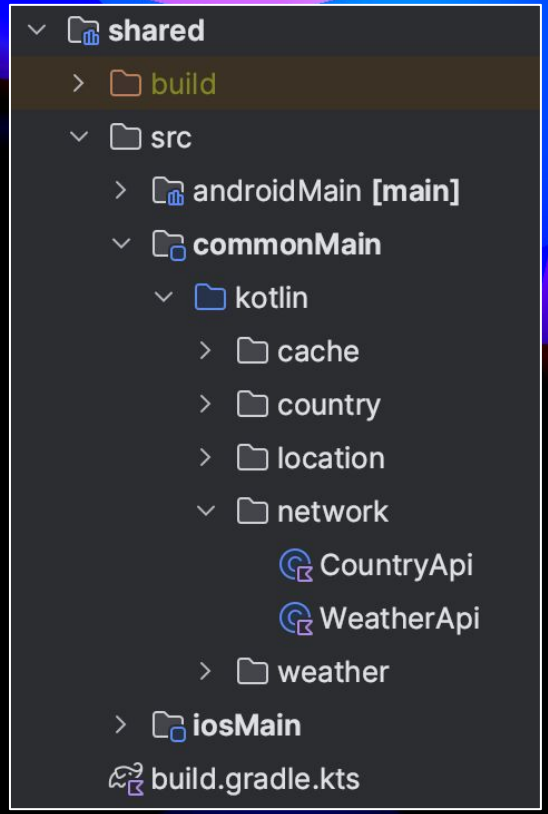
# Types to support networking

Our networking code uses two servers:

- restcountries.com for countries
- api.openweathermap.org for weather

We have a client for each one:

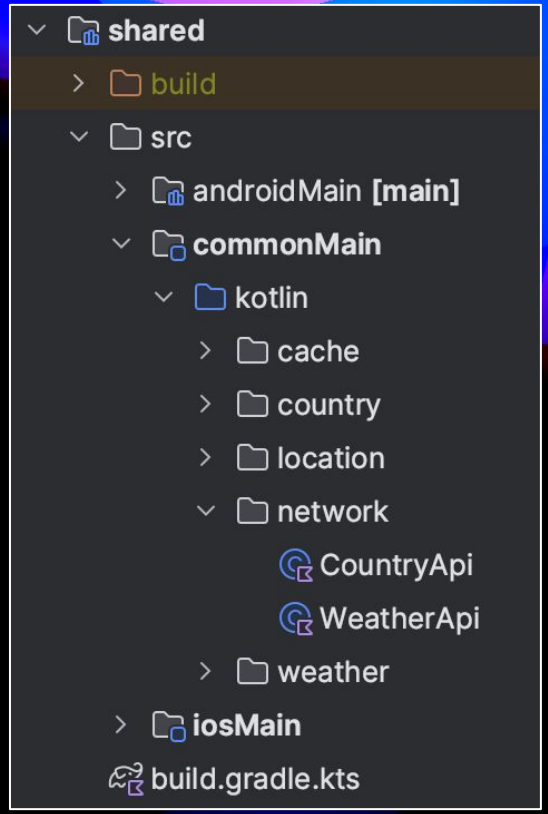
- CountryApi
- WeatherApi



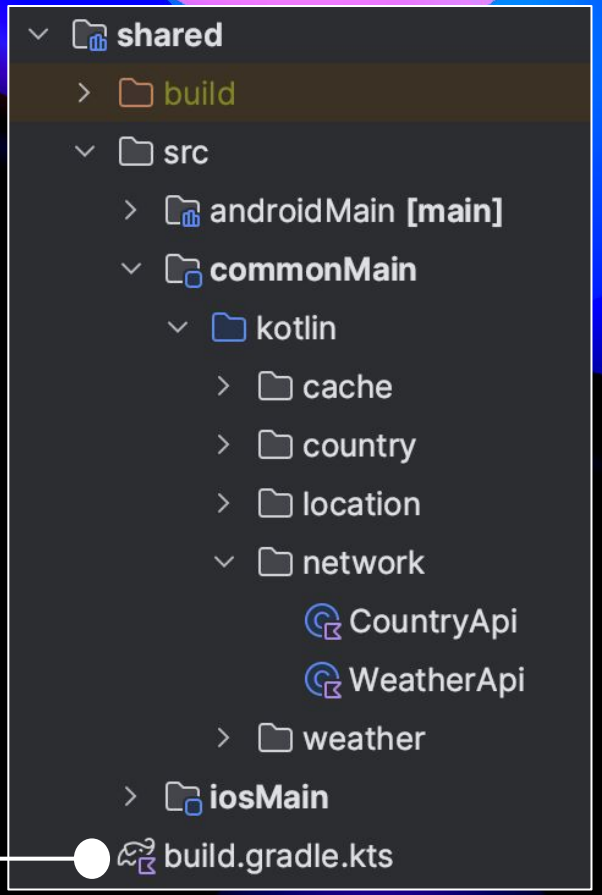
# Types to support networking

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 {
        ...
    }
}
```



```
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
)
```

```kotlin
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()
}
```

```kotlin
@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

## 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.serialisation 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

```
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

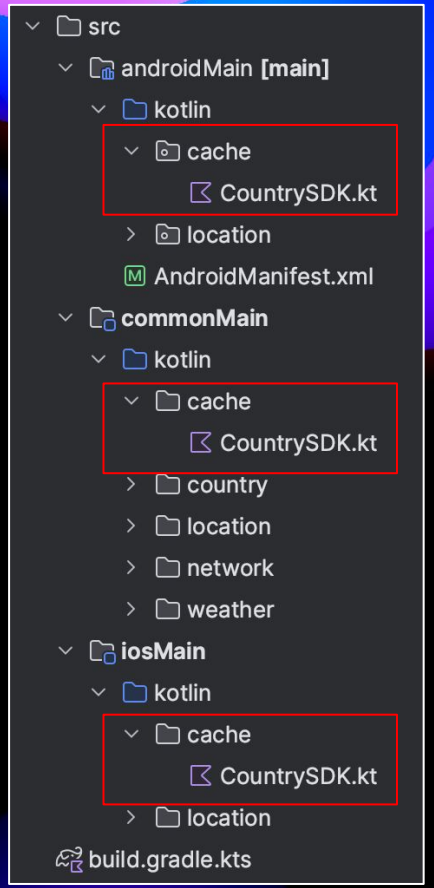
# Platform specific types for caching

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



```kotlin
//In shared/src/commonMain/kotlin/cache

expect fun pathToCountryCache(): String
```

```kotlin
//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"
    }
}
```

```kotlin
//In shared/src/commonMain/kotlin/cache

expect fun pathToCountryCache(): String
```

```kotlin
//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

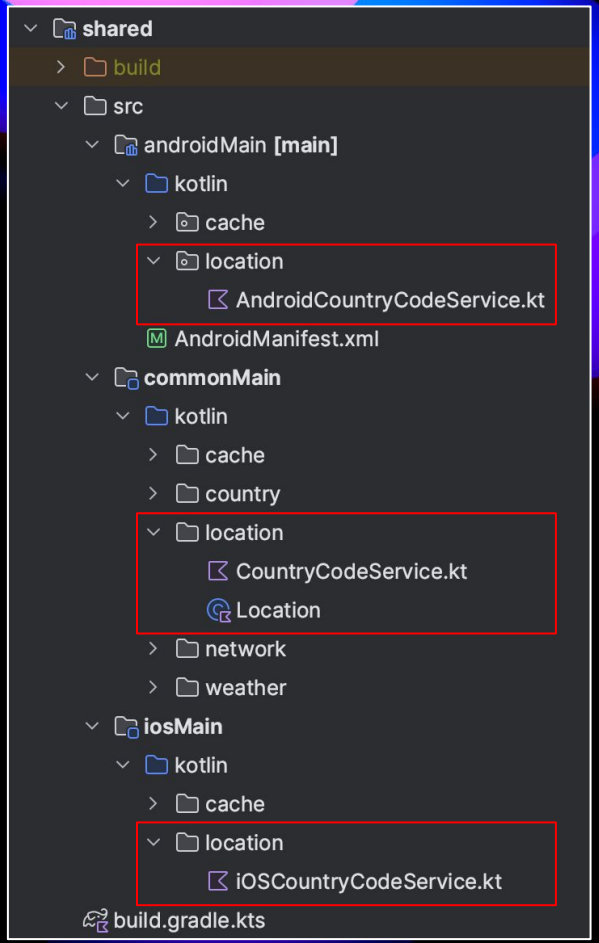
Data in interfaces

# 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



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

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

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

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

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

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

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

```kotlin
//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

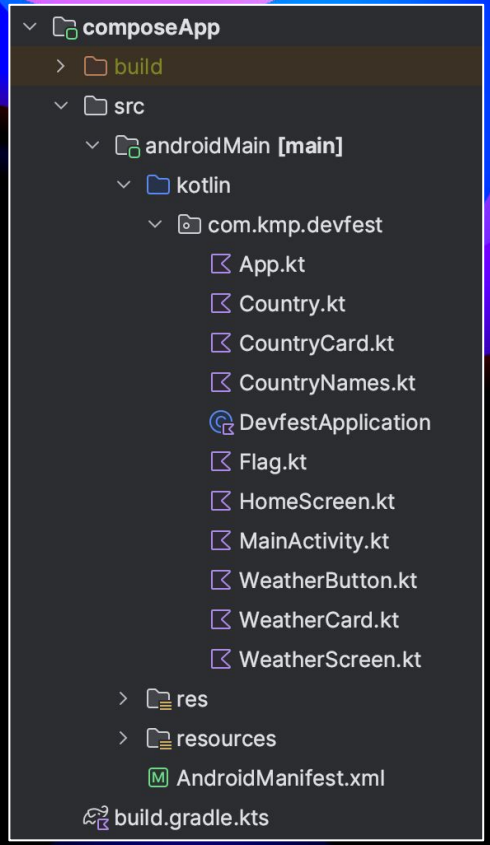
Our Android UI uses Jetpack Compose

- It is made up of Composable Functions

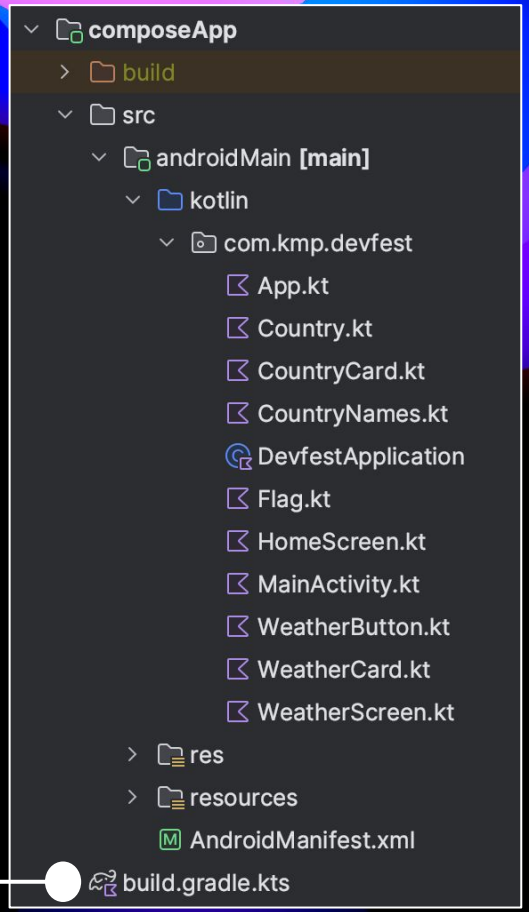
These use Android specific libraries

- 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

```
@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

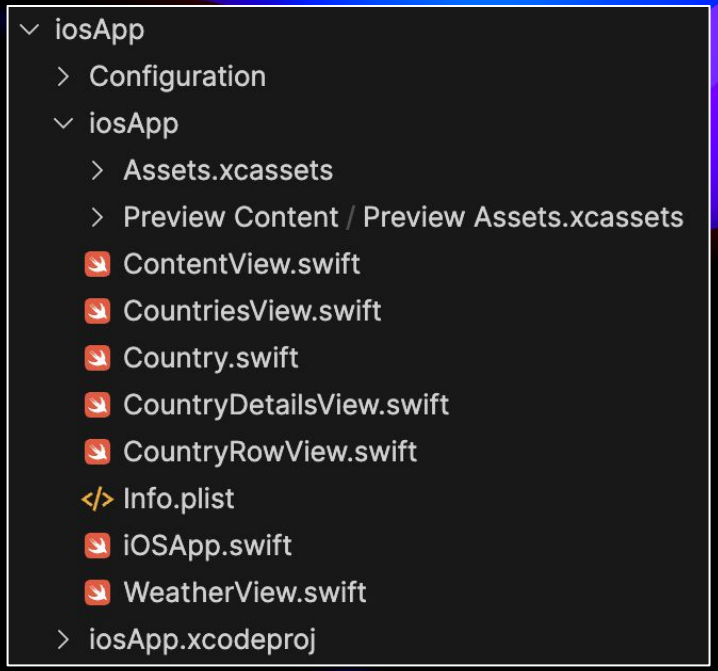
We create structures which inherit from View

Then arrange then 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



# The CountryDetailsView

VStack with two Text views

KFImage

```swift
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 ✓

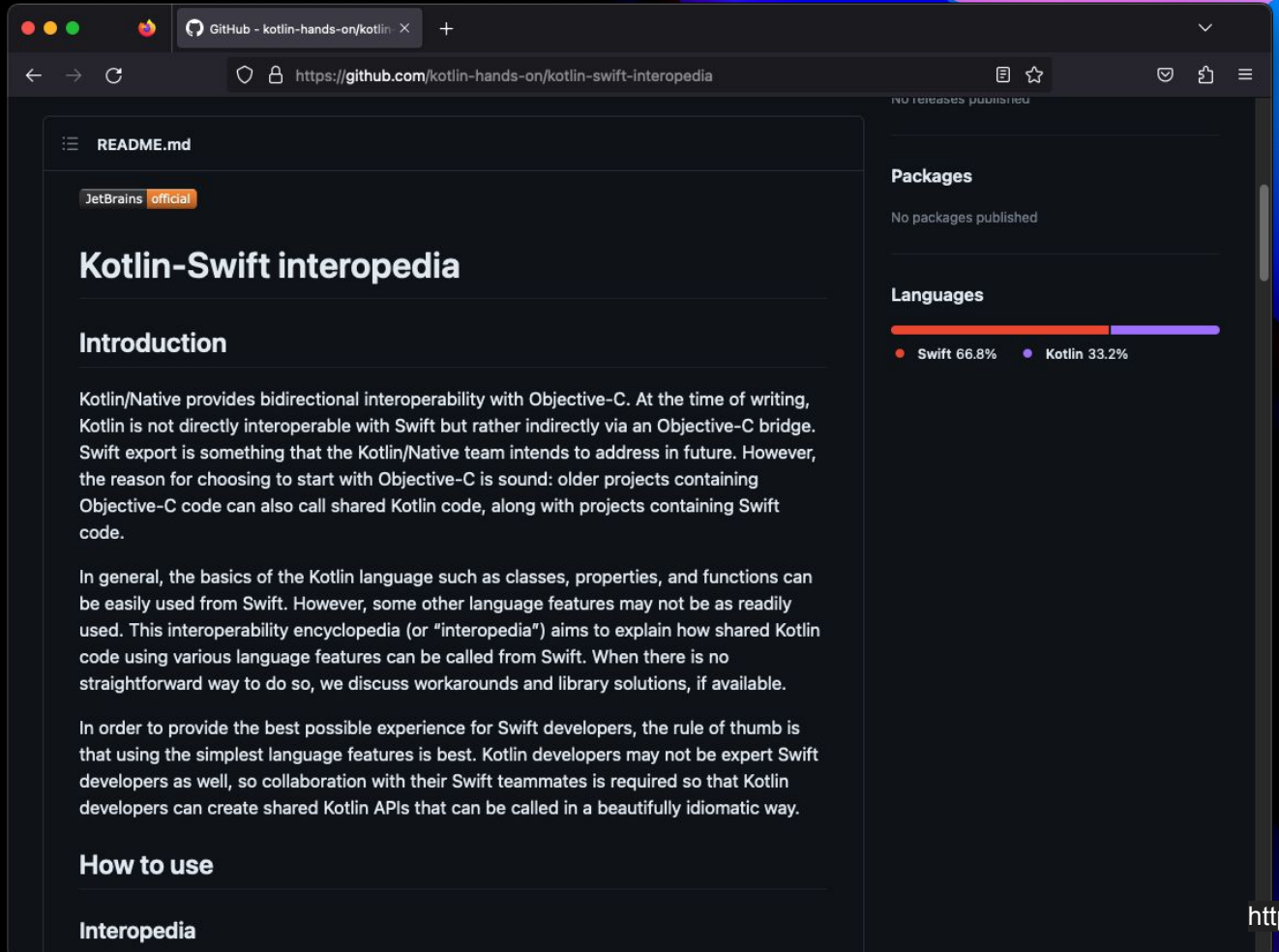
Platform specific support:

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

A JetPack Compose based interface ✓

A SwiftUI based interface ✓

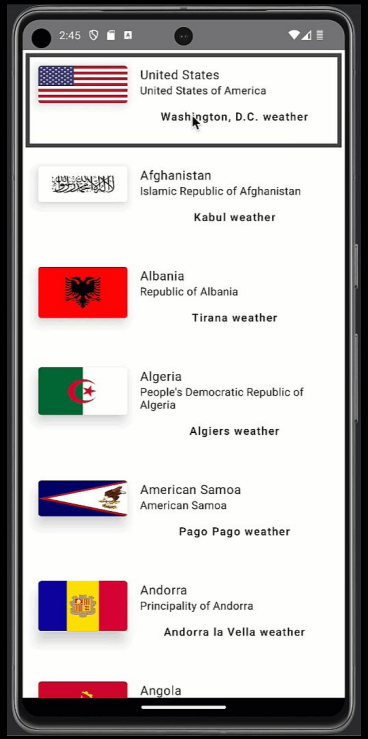
Data in interfaces ✓



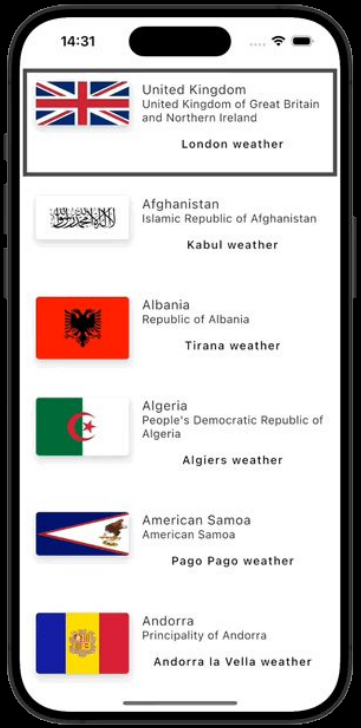

https://kotl.in/interopedia

# Building an App
(with Shared UI)

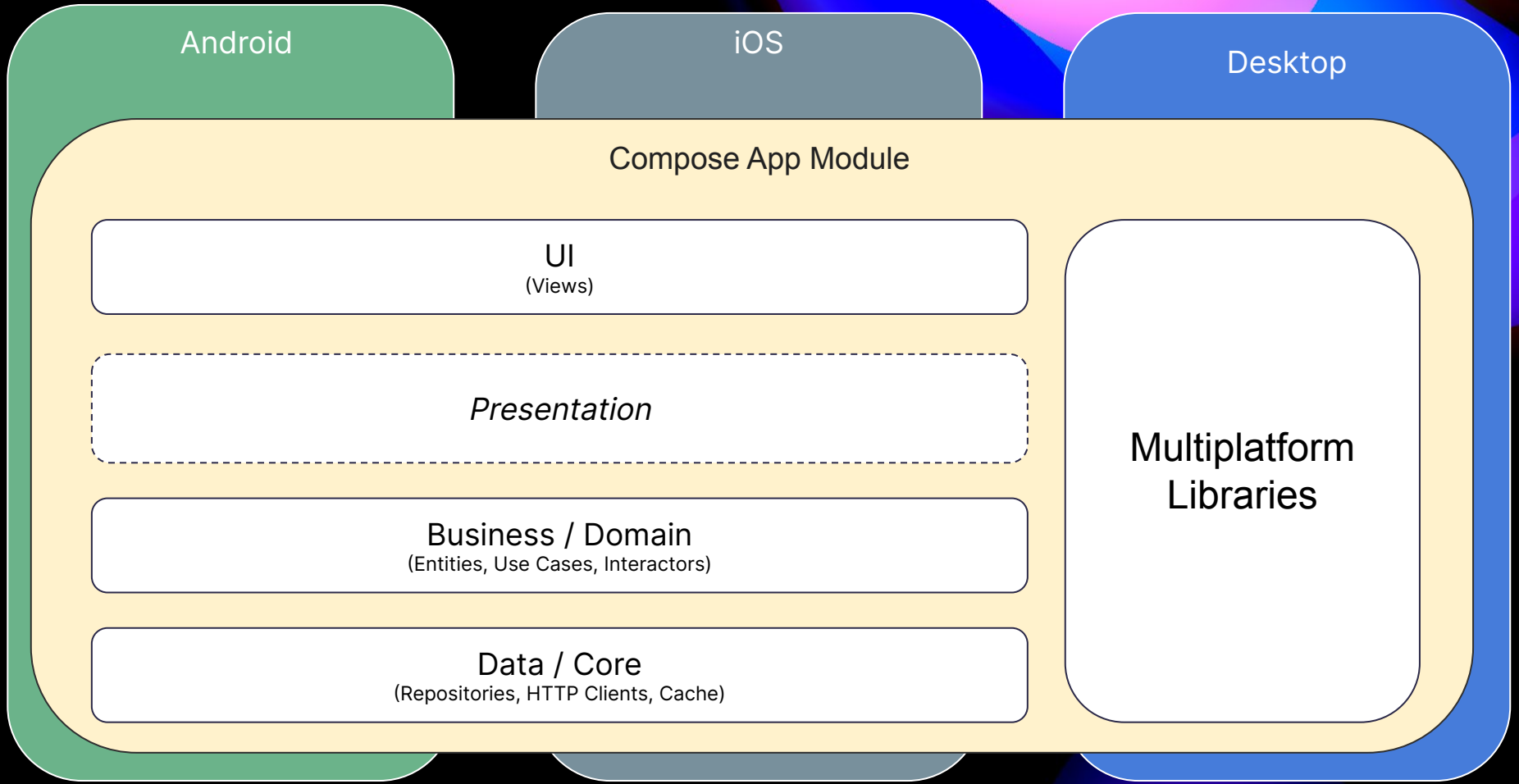
Android + CM




iOS + CM




Desktop + CM

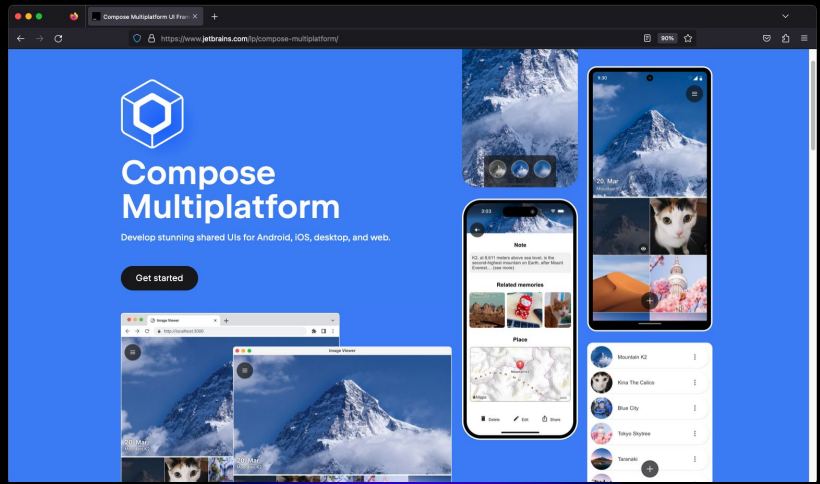

Android
iOS
Desktop
Compose App Module
UI
(Views)
Presentation
Business / Domain
(Entities, Use Cases, Interactors)
Data / Core
(Repositories, HTTP Clients, Cache)
Multiplatform Libraries


This will be eas(y|ier)

# Moving to a shared UI

Our UI will now be implemented as common code

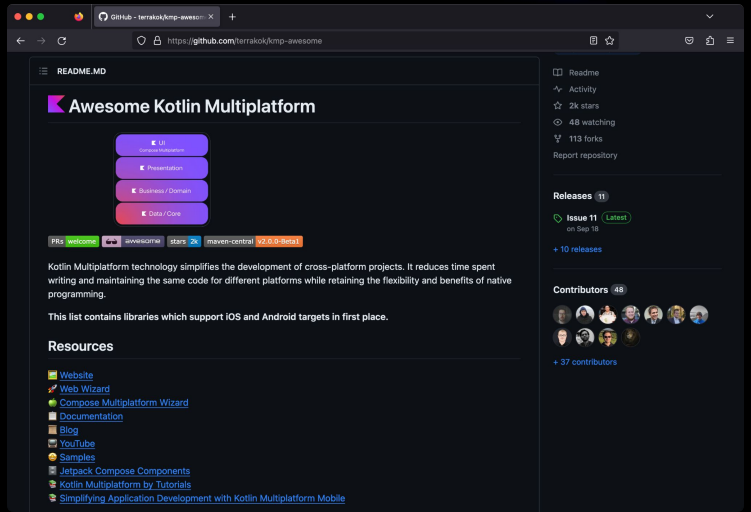
- The SwiftUI version is no longer required
- The Jetpack Compose version is ported

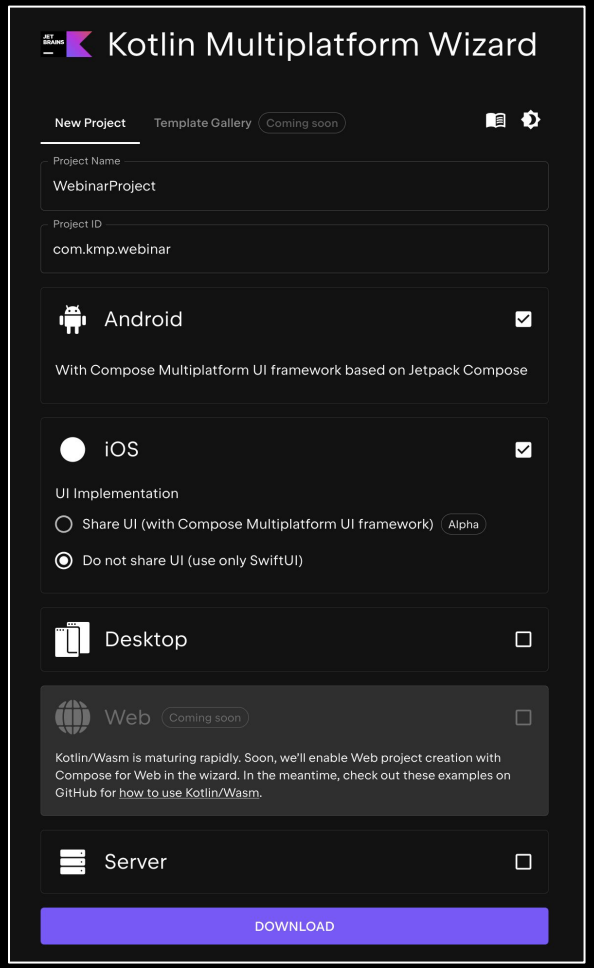


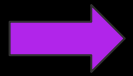
# Moving to a shared UI

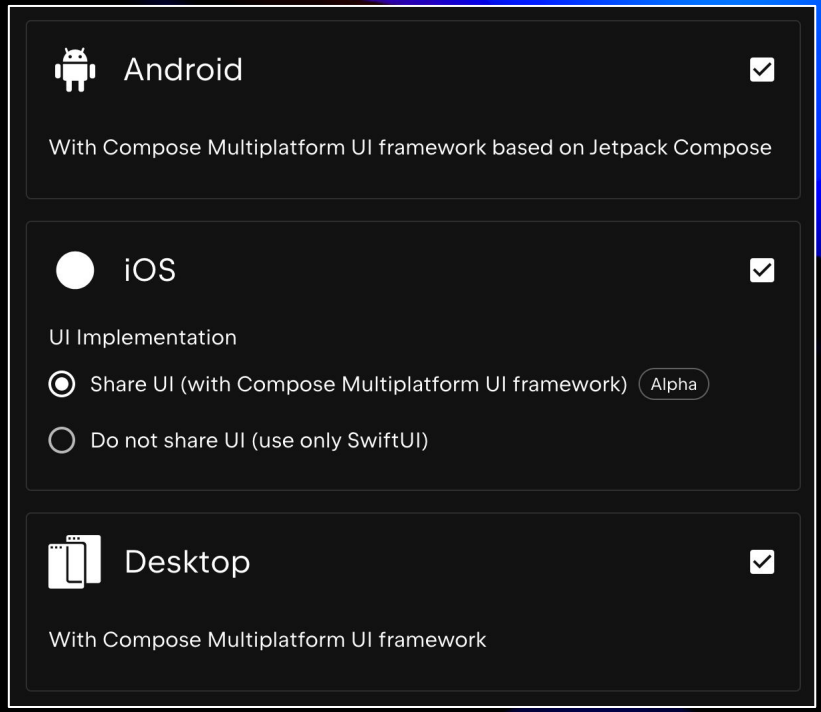
To move from Jetpack Compose to Compose Multiplatform

- 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

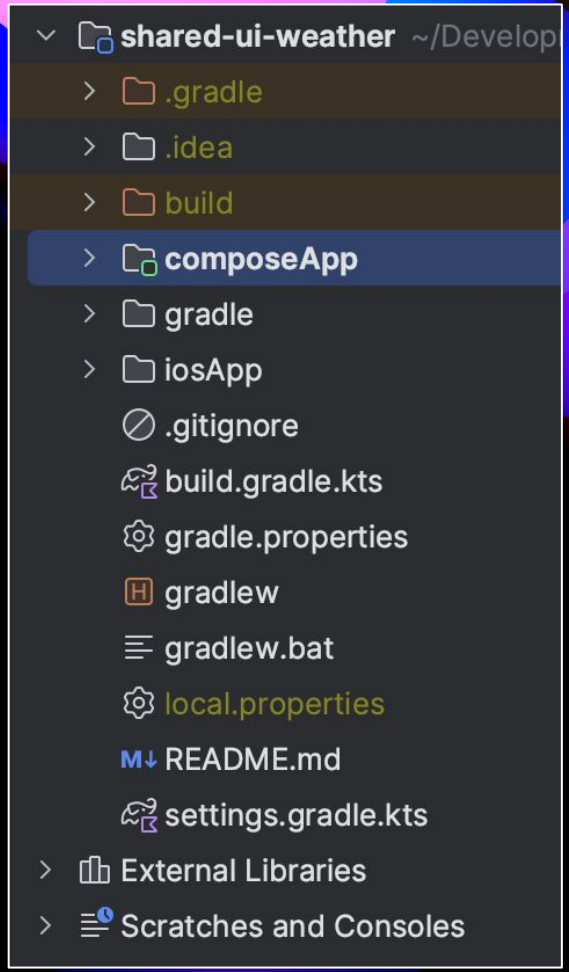
# The project structure

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

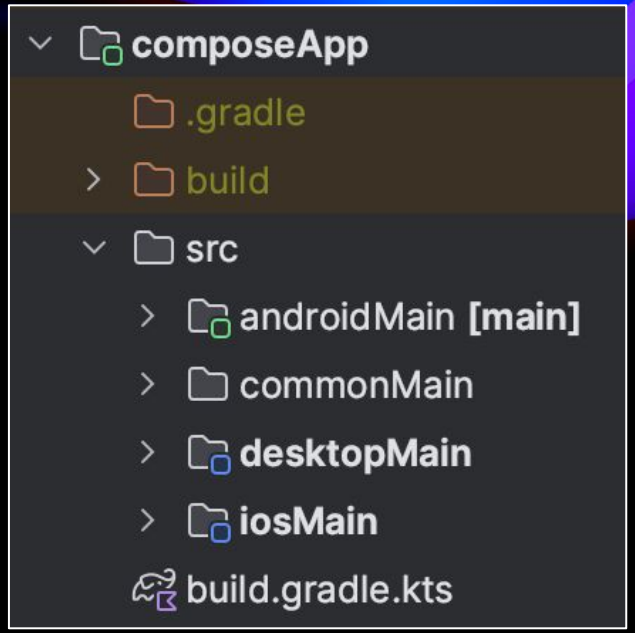


# The composeApp module

The module contains four source sets

A commonMain source set for shared code

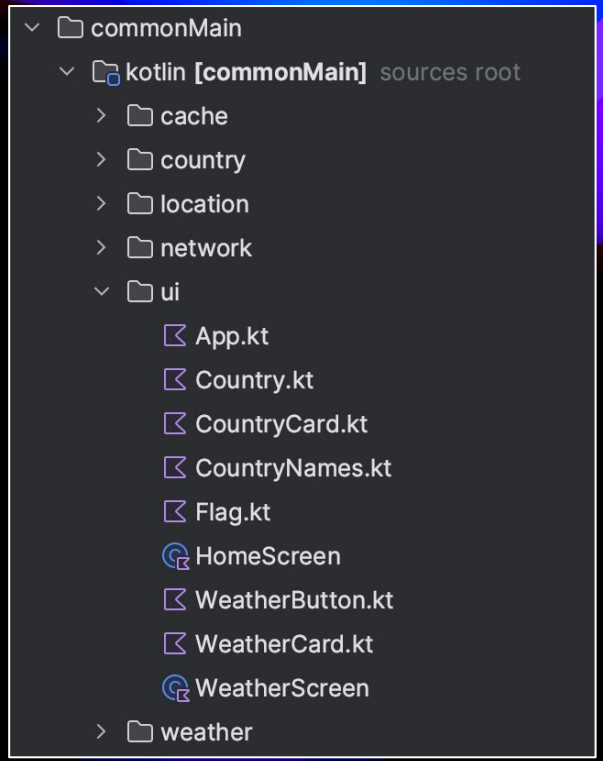
One source set for each targeted platform



# The commonMain source set

The shared UI lives inside commonMain

We create a new package for our composables



# 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

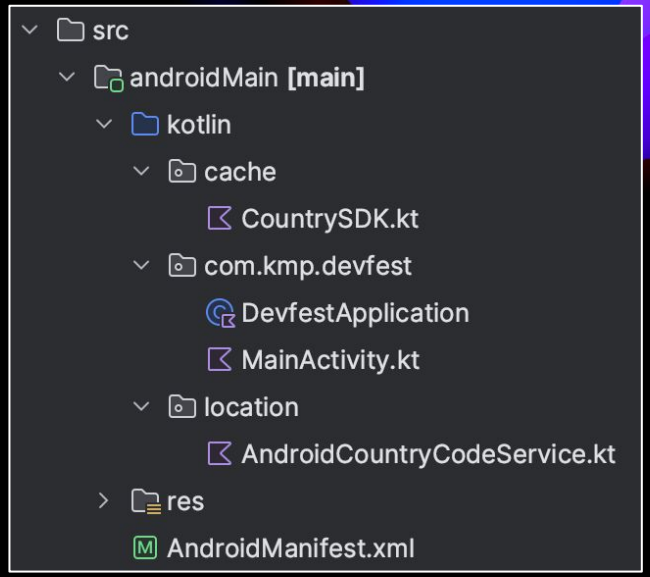
```
androidMain.dependencies {
    ...

  implementation("io.coil-kt:coil-compose:2.5.0")
  implementation("androidx.navigation:navigation-compose:2.7.5")

}

commonMain.dependencies {
    ...
  api("io.github.qdsfdhvh: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()
        }
    }
}
```

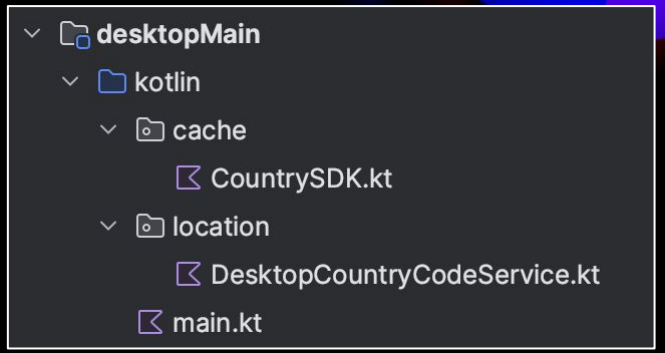
# The androidMain source set

```
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



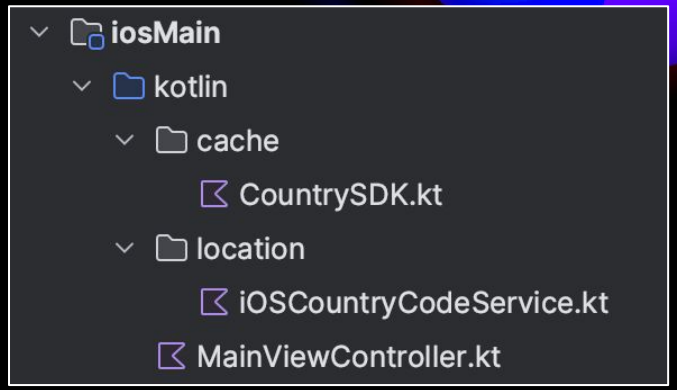
# The desktopMain source set

```
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

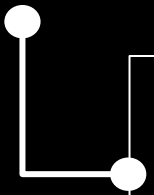
You have access to an ecosystem of multiplatform libraries

- 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]
```

# Thank you 

garth.gilmour@jetbrains.com
@GarthGilmour