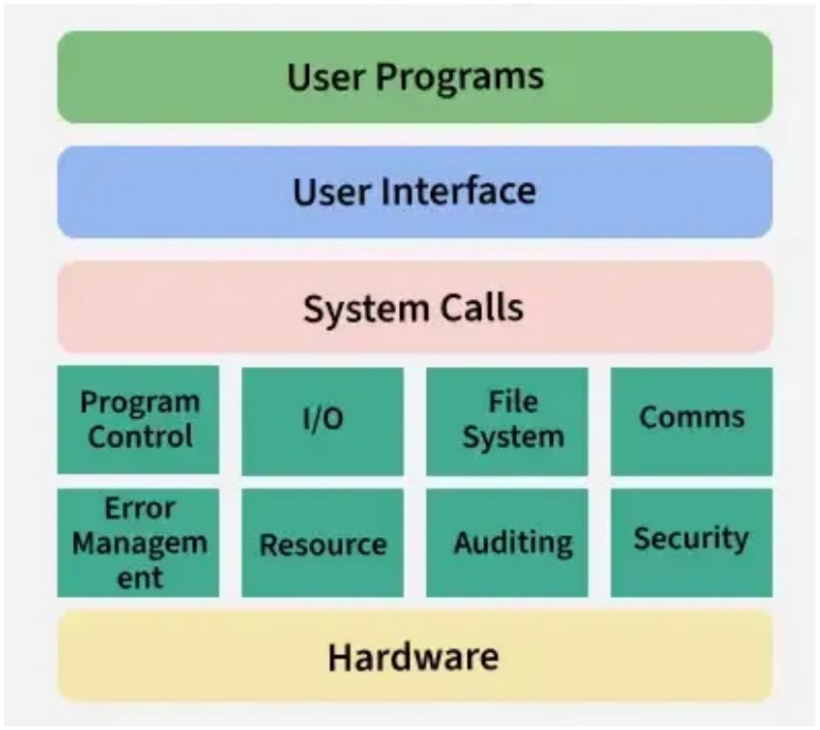
Kotlin Multiplatform

CS 346 Application Development

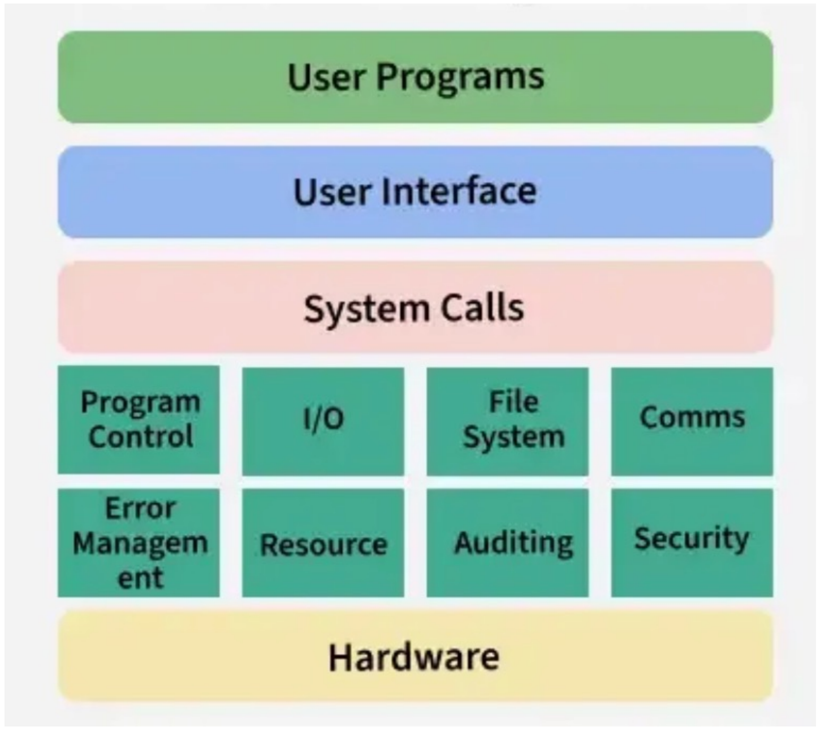
Native compilation

- Many programming languages use **native compilation**: translating program instructions into a low-level instruction format that be executed on specific hardware.
  - Libraries in this pipeline need to be native to that platform!
  - Output is a compiled executable that the OS can interpret/run.
- OS vendors provide access to the underlying OS though system libraries e.g., graphics, networking.
  - Your application is relying heavily on system calls for complex functionality.



The challenge

- Native compilation and using the underlying OS libraries as much as possible will give you the best performance.
  - Any third-party libraries are effectively abstractions over the OS.
  - The OS itself controls execution.
- If you want to support multiple targets, you need to rebuild your application for each platform.
  - Few cross-platform standards exist.
  - e.g., OpenGL, DirectX.



Building the same application multiple times, for each platform, is expensive!!

Cross-platform development

1990-2005

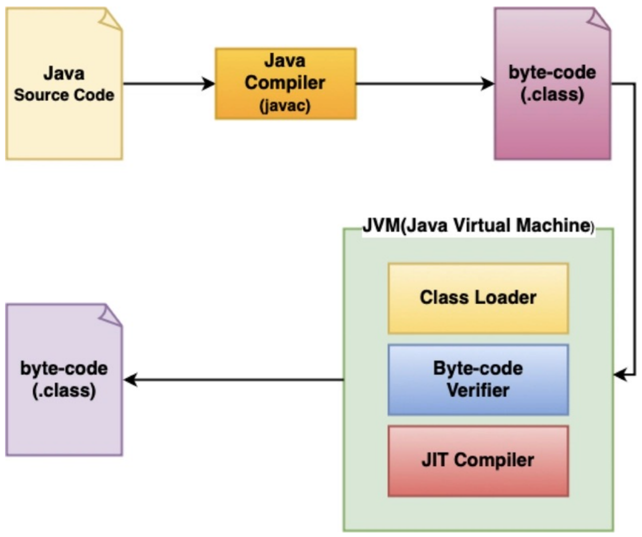
- Desktop was dominated by Windows.
- Workstations running Solaris, HPUX were rare.
- Servers were often commercial *nix.

1996

- Java attempted a coup.
- Compile to an intermediate format and then provide a VM that can run on each target.
- “Write once, run anywhere”.
- Worked great for backends, not-so-much front-end.

2000s

- Cross-platform libraries exist but are difficult to develop and maintain; also tend to not work as well as native.



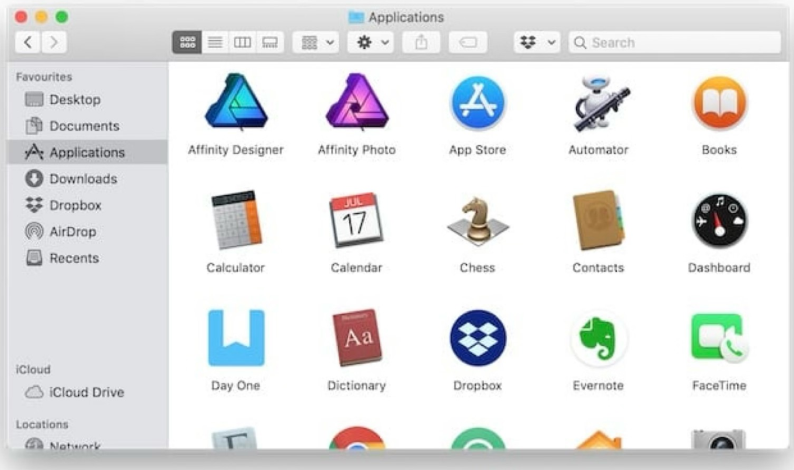
https://dev.to/binoy123

Today's environment

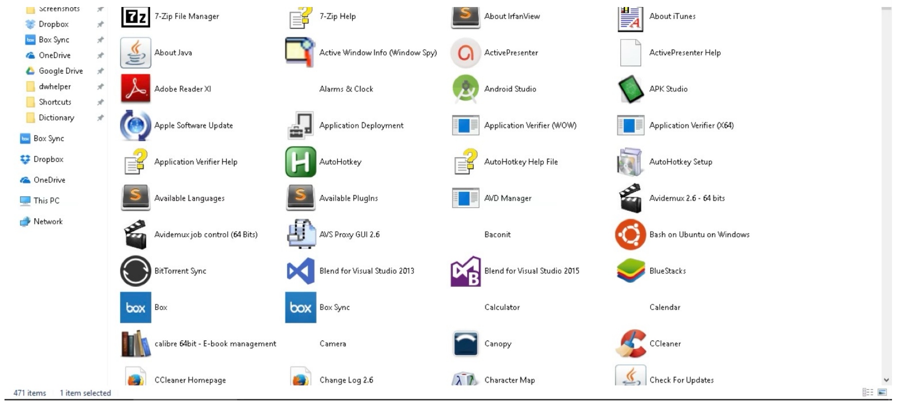
iOS

watchOS

Android



Mac



Windows

Web applications dominate

- Cheaper to produce, easy to deploy, but resource intensive. Often a worse-version of an application.
- Bundling JS + chrome is a poor technical choice.

Native applications are preferable, but also expensive

- Can we make cross-platform viable?

Comparison

Native applications

- Architecture-specific tooling usually provided by vendors
  - Best user experience
  - Great performance
  - Best way to leverage OS features.
  - **Cannot reuse code across platforms.**
- Examples
  - C# and .NET on Windows
  - Swift on macOS.

$\longleftrightarrow$

Decision

Cross-platform applications

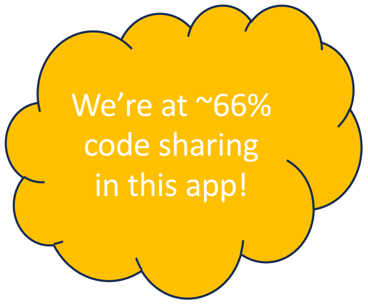
- Tooling developed for multiple platforms
  - Compromised user experience ("worst of both worlds")
  - Performance may be compromised
  - May not have complete access to underlying functionality.
  - **CAN reuse code across platforms.**
- Examples
  - C++, Qt is somewhat portable
  - Java, Swing

Code sharing in "*courses*"

Console, desktop and Android versions from one repo!
Each represents is a different compiler target.

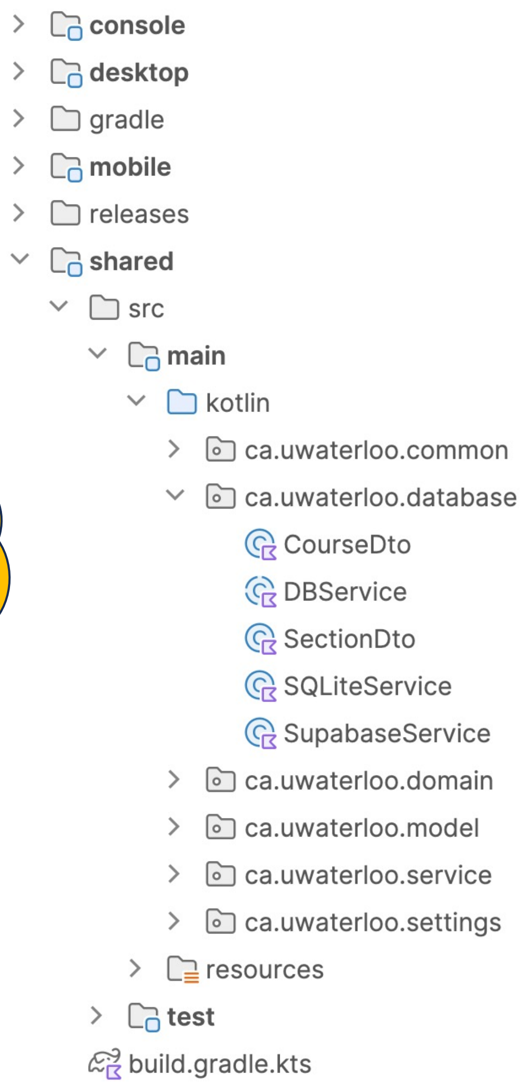
We were able to share:

- Web service code
- Database access code
- Model and domain objects.



UI code is platform specific and <u>cannot</u> be shared/reused.

- Console uses stdlib.
- Desktop uses Compose Multiplatform.
- Mobile uses Jetpack Compose.



```
> console
> desktop
> gradle
> mobile
> releases
v shared
  v src
    v main
      v kotlin
        > ca.uwaterloo.common
        v ca.uwaterloo.database
            CourseDto
            DBService
            SectionDto
            SQLiteService
            SupabaseService
        > ca.uwaterloo.domain
        > ca.uwaterloo.model
        > ca.uwaterloo.service
        > ca.uwaterloo.settings
      > resources
    > test
  build.gradle.kts
```

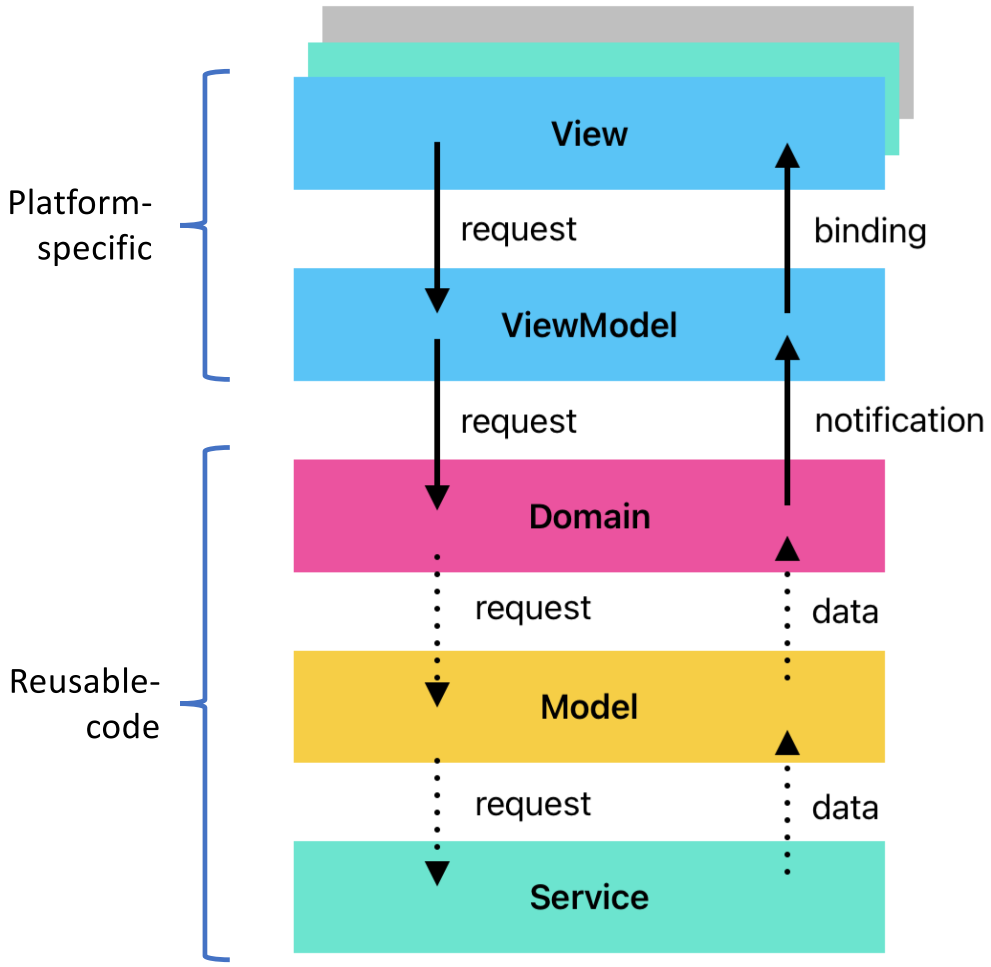
Limits of code sharing

This is about as far as we can normally go when trying to code share

- Not reusable (~1/3).
  - User interface
  - Networking, DB
- Reusable (~2/3).
  - Domain
  - Model
  - Portable libraries

This is actually pretty good…

We'd like to get to 100% reusable.



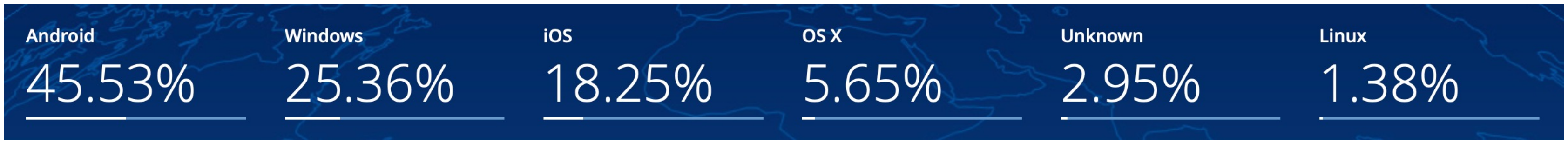
Kotlin Multiplatform

How to make your business logic, back-end code portable.

Good Enough?

We can go *pretty-far* with code reuse - assuming a programming language and required libraries exist across all platforms.

The challenge is that our two most important platforms are both GUI-based are are completely incompatible with one another.

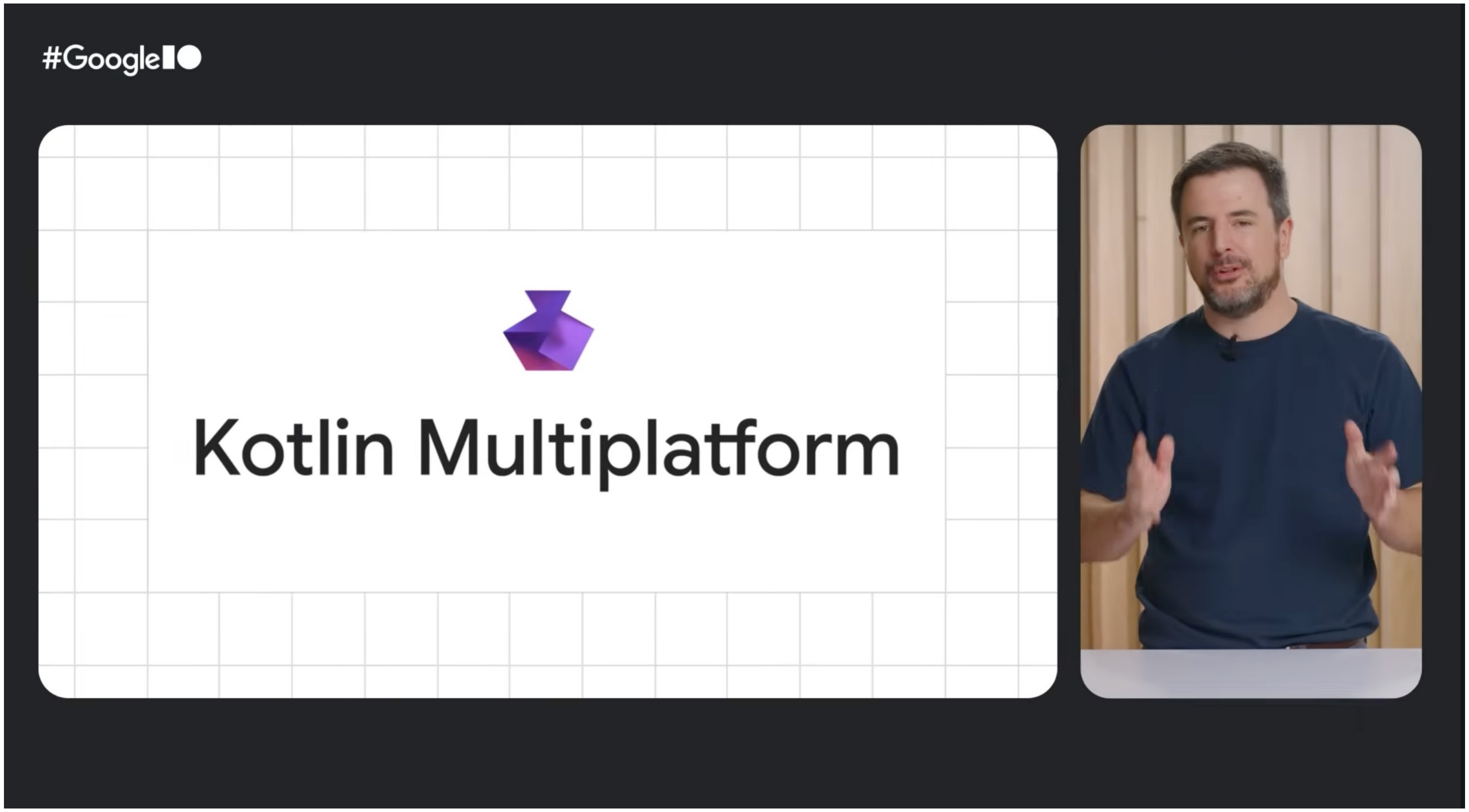


Android	Windows	iOS	OS X	Unknown	Linux
45.53%	25.36%	18.25%	5.65%	2.95%	1.38%

Market share by Operating System (2025)

- **Android**: Kotlin + Jetpack Compose native
- **iOS**: Swift, Swift UI (Carbon) native

} Well. Now what?



https://youtu.be/gP5Y-ct6QXI?t=14

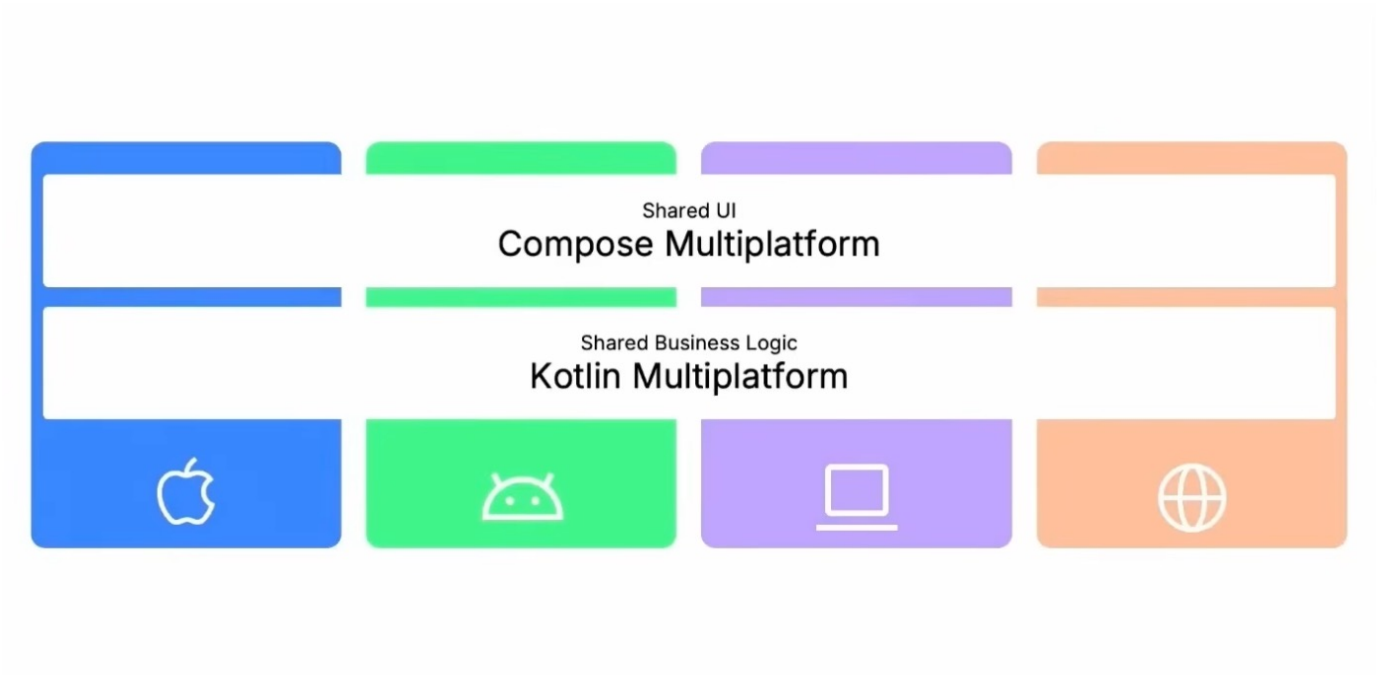
KMP

"Kotlin Multiplatform (KMP) is a technology that allows developers to write code once and deploy it across multiple platforms, like Android, iOS, desktop, and web, using a single Kotlin codebase. "

- https://kotlinlang.org/docs/multiplatform.html

What makes it different?

- Native compilation for multiple platforms
- You can mix shared Kotlin code with native OS/toolkit calls.
- Build desktop, Android, iOS, Web, WASM from a single Kotlin code-base.
    - Reduced development costs, reduced code duplication, reduced time-to-market.



KMP lets us mix shared and platform-specific code for all these platforms: iOS, Android, desktop/JVM (Linux, macOS, Windows), Web.

We can also include libraries that meet these conditions:

- KMP libraries - target any of these platforms
- C/C++ libraries – using Kotlin interop.
- iOS/mac libraries – using Kotlin interop with Swift and Objective C.

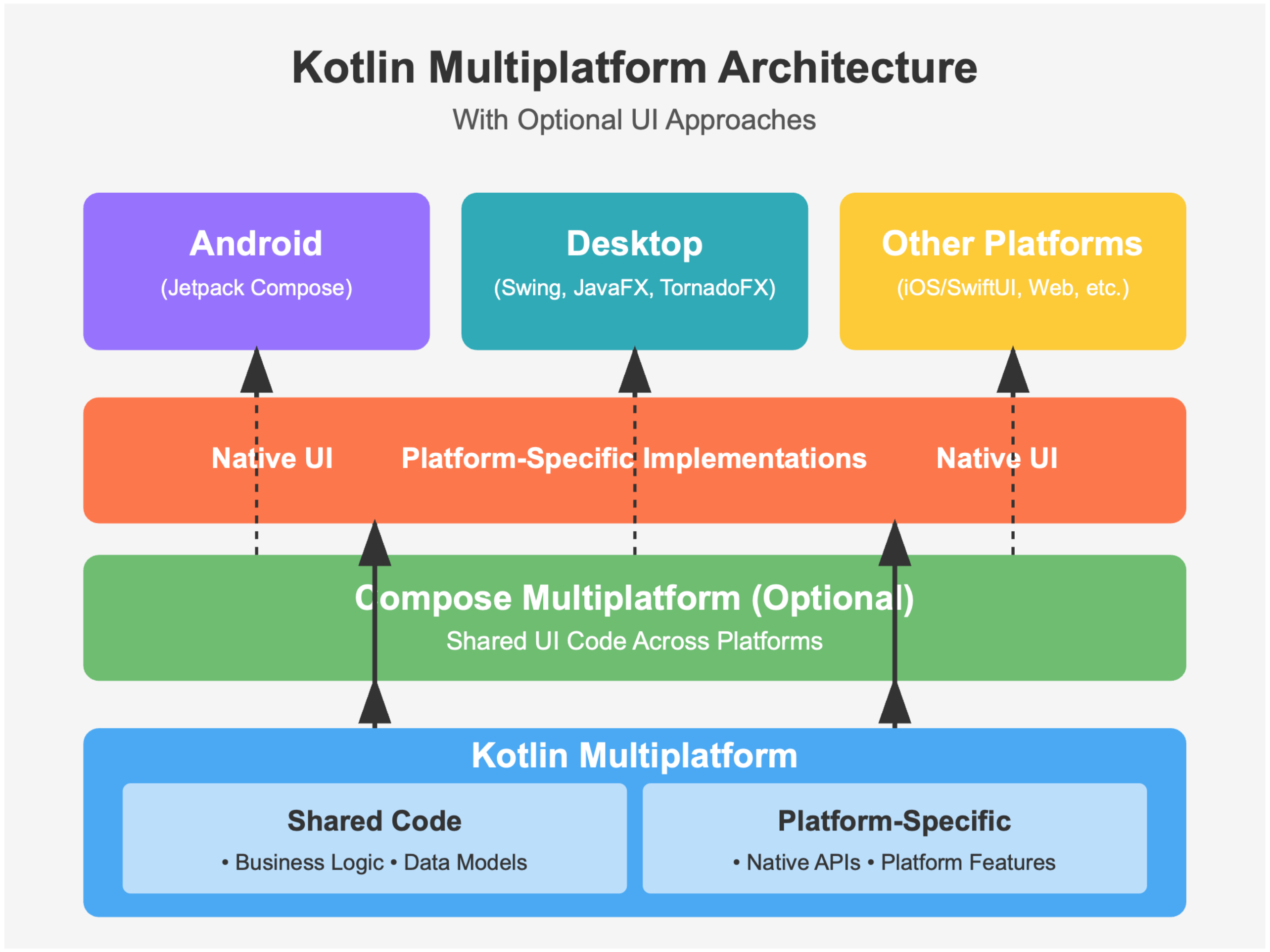
Supported Use Cases for KMP

1. Share a piece of logic:
   - Create a (non-UI) module in Kotlin and export as a Swift-compatible library.
2. Share all logic with a native UI for each platform:
   - 66% of your application in written in Kotlin.
   - 33% is native code (e.g., Compose for Android, Swift/UI for iOS).

Needs a way to make native calls from Kotlin.

- Share all logic with shared UI for most/all platforms:
  - Build 100% of your application in Kotlin, using Kotlin toolchain.
  - Integration-point is an Xcode project that Kotlin tools generate.
    - Gradle builds Android project
    - Xcode builds iOS project

Needs to support both toolchains (compilation, code-signing).


Kotlin Multiplatform Architecture
With Optional UI Approaches
Android
(Jetpack Compose)
Desktop
(Swing, JavaFX, TornadoFX)
Other Platforms
(iOS/SwiftUI, Web, etc.)
Native UI
Platform-Specific Implementations
Native UI
Compose Multiplatform (Optional)
Shared UI Code Across Platforms
Kotlin Multiplatform
Shared Code
• Business Logic • Data Models
Platform-Specific
• Native APIs • Platform Features

How is this possible?!

It’s a combination of:

- Compilers for every platform.
    - JVM compiler for desktop
    - Native compilers for iOS, Android, WASM
- Libraries that run native across multiple platforms – see Klibs.io
    - Coil for imaging, Ktor for networking, ,…
    - Compose Multiplatform for UI was critical for a cross-platform UI story!
- Tooling to integrate any platform with any other platform’s code.

Restrictions?

- You need cross-platform libraries OR you need to dip into native calls.
- Similar approach taken by competitors (Flutter/Dart, React Native).

Setting up a KMP project

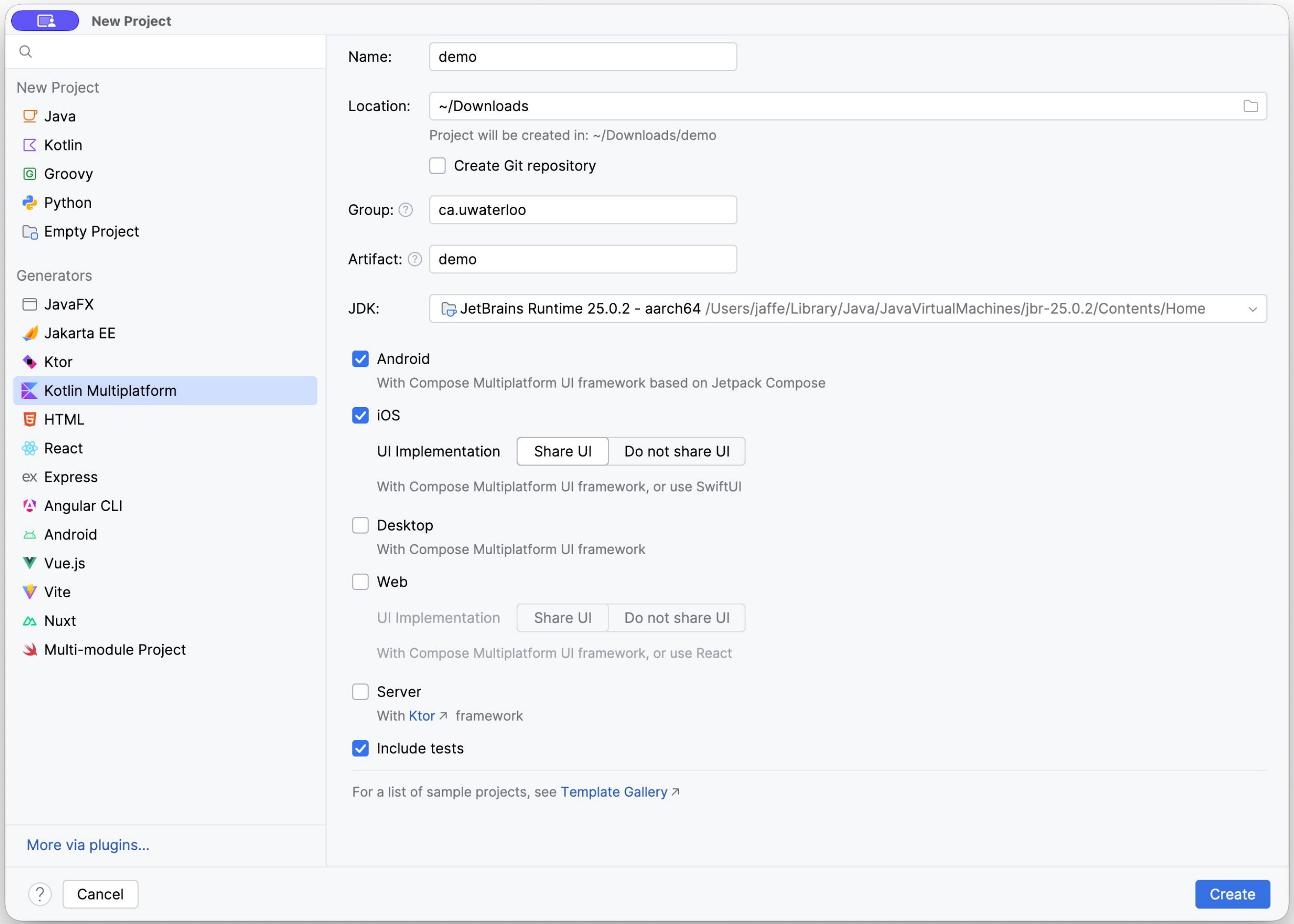
How to get started

Requirements

1. IntelliJ IDEA 2.25.2.2 or Android Studio Otter 2025.2.1.
   - IDEA is more “cutting edge”
2. Kotlin Multiplatform IDE plugin.
   - Works in either IDE, for all platforms.
3. Xcode on a Mac for iOS development.
   - Make sure that you run Xcode first, accept license agreements.

Note that parts of the platform are still in development.

- Desktop, Android, iOS – 100% stable.
- WASM, web? – beta
- Amper (replacement for Gradle)? - alpha



New Project
- Java
- Kotlin
- Groovy
- Python
- Empty Project

Generators
- JavaFX
- Jakarta EE
- Ktor
- Kotlin Multiplatform
- HTML
- React
- Express
- Angular CLI
- Android
- Vue.js
- Vite
- Nuxt
- Multi-module Project

More via plugins...

Name: demo

Location: ~/Downloads

Project will be created in: ~/Downloads/demo

[ ] Create Git repository

Group: ca.uwaterloo

Artifact: demo

JDK: JetBrains Runtime 25.0.2 - aarch64 /Users/jaffe/Library/Java/JavaVirtualMachines/jbr-25.0.2/Contents/Home

[✓] Android
With Compose Multiplatform UI framework based on Jetpack Compose

[✓] iOS
UI Implementation [✓ Share UI] Do not share UI
With Compose Multiplatform UI framework, or use SwiftUI

[ ] Desktop
With Compose Multiplatform UI framework

[ ] Web
UI Implementation Share UI Do not share UI
With Compose Multiplatform UI framework, or use React

[ ] Server
With Ktor ↗ framework

[✓] Include tests

For a list of sample projects, see Template Gallery ↗

New > Project, with the Kotlin Multiplatform plugin installed.

Project Structure

DEMO

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

composeApp includes all Compose code. It is further split into android, common, desktop and iOS.

- This is where you add source code.

iosApp includes the iOS project and configuration files, used to build and package using Xcode and other macOS tools.

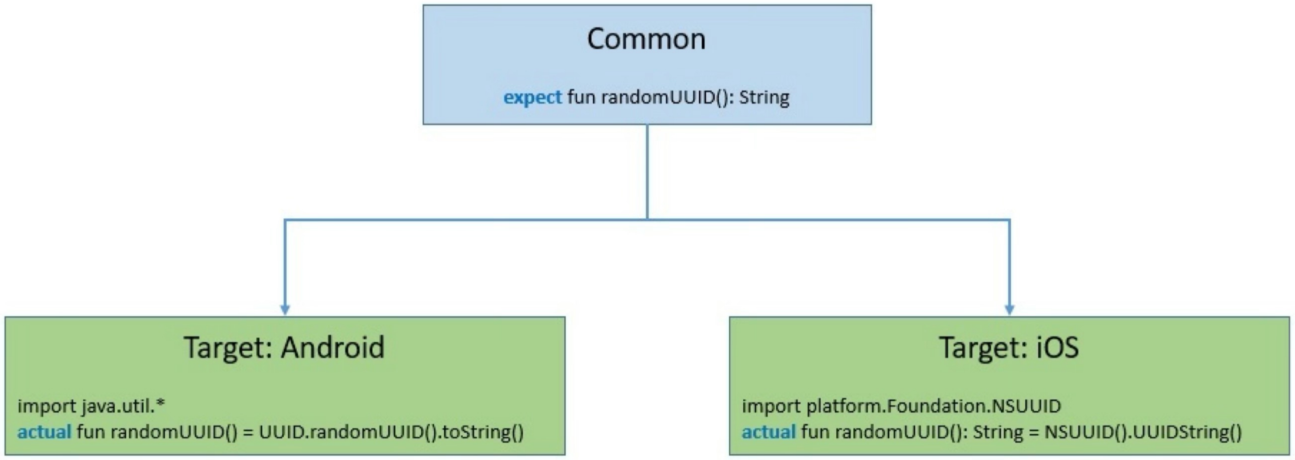
- Integration point for Kotlin/iOS.
- You probably shouldn’t touch this.

```
.
├── build.gradle.kts
├── composeApp
│   ├── build.gradle.kts
│   └── src
│       ├── androidMain
│       ├── commonMain
│       ├── desktopMain
│       └── iosMain
├── gradle
├── gradle.properties
├── iosApp
│   ├── Configuration
│   │   └── Config.xcconfig
│   ├── iosApp
│   │   ├── Assets.xcassets
│   │   ├── ContentView.swift
│   │   ├── Info.plist
│   │   ├── Preview Content
│   │   └── iOSApp.swift
│   ├── iosApp.xcodeproj
│   └── project.pbxproj
├── local.properties
└── settings.gradle.kts
```

Calling native code

KMP add two new keywords:

- **expected** to indicate a required function in common modules,
- **actual** declarations in the platform specific modules.



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

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

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

Example: common code

/commonMain
Just regular Kotlin code.

```
fun add(num1: Double, num2: Double): Double {
    val sum = num1 + num2
    writeLogMessage("The sum of $num1 & $num2 is $sum", LogLevel.DEBUG)
    return sum
}


fun subtract(num1: Double, num2: Double): Double {
    val diff = num1 - num2
    writeLogMessage("The difference of $num1 & $num2 is $diff", LogLevel.DEBUG)
    return diff
}
```

Example: platform specific

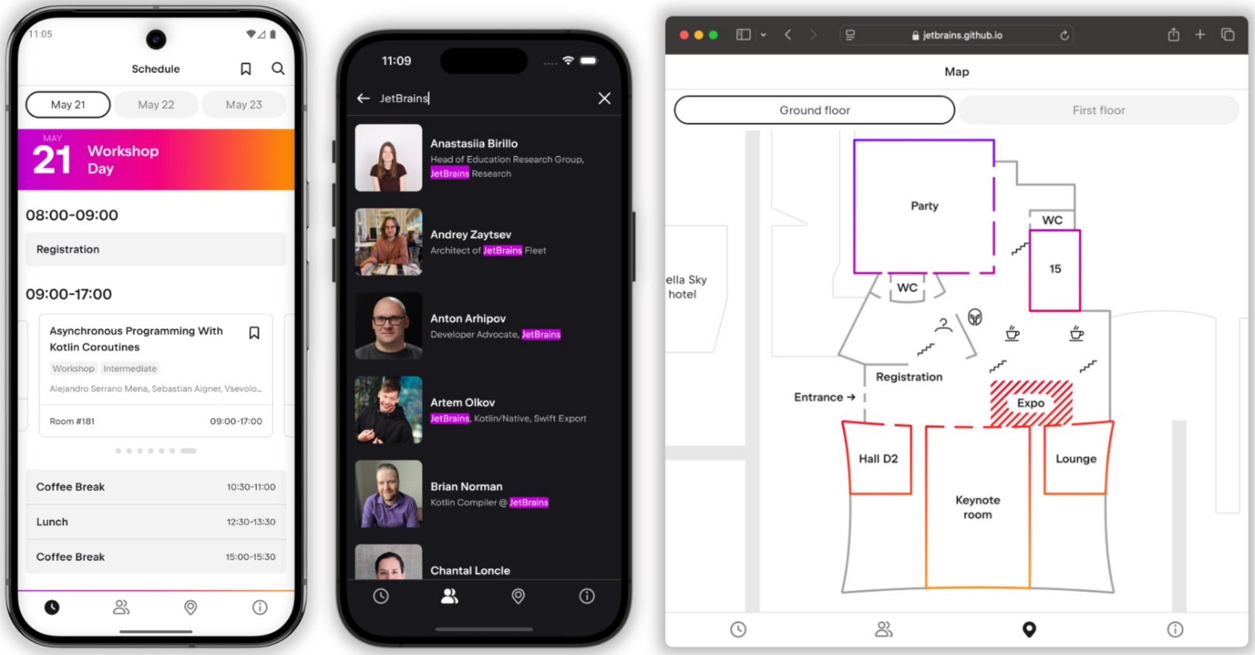
/desktopMain
/androidMain
/iosMain

The `writeLogMessage()` function should be platform specific, since each OS will handle this differently. We will add a top-level declaration to our **common** module defining the function:

```
enum class LogLevel {
    DEBUG, WARN, ERROR
}
internal expect fun writeLogMessage(message: String, logLevel: LogLevel)
```

The `expect` keyword tells the compiler that the definition will be in a **platform** module.

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("Running in JVM: [$logLevel]: $message") // desktopMain
}
```



DEMO

There are many sample KMP applications, and examples of commercial applications that use it.
This is the [KotlinConf conference guide](https://kotlinlang.org) application, which works on all platforms!

https://kotlinlang.org/docs/multiplatform/multiplatform-samples.html#jetbrains-official-samples

Reference

JetBrains. 2025. KotlinConf Talks.

JetBrains. 2025. Kotlin Multiplatform Documentation

Moore, Mota, Taheri. 2025. Kotlin Multiplatform by Tutorials. ⭐

Lackner. 2024. KMP vs. Flutter - Who Will Win The Cross-Platform Battle?.