KMP

for Mobile Native Developers



A Hands-on Guide to **Kotlin Multiplatform** for Android and iOS

SANTIAGO MATTIAUDA

KMP for Mobile Native Developers

A Hands-on Guide to Kotlin Multiplatform for Android and iOS



Leanpub

First published by Leanpub 2026

Copyright © 2026 by Santiago Mattiauda

All rights reserved. No part of this publication may be reproduced, stored, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.

Second edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

Preface viii
Acknowledgments x
1 What is cross-platform mobile development? 1
Understanding what "multiplatform" means in practice 2
Choosing an approach for Android and iOS (and understanding its trade-offs) 3
Making a practical decision: native vs. code sharing 7
Assess whether the multiplatform approach is right for you 8
Understanding the landscape: Flutter, React Native, and Kotlin Multiplatform 9
Summary 13
Verification checklist 13
In the next chapter 14
2 Introduction to Kotlin Multiplatform 15
What is Kotlin Multiplatform (KMP)? 16
Adoption strategies (from smallest to largest scope) 20
How to extract real value from shared code 26
A flexible, non-exclusive model 27
Why adopt KMP (benefits and limitations) 29
How to present it to your team (practical framework) 29
Summary 31
Verification checklist (before moving forward) 31
Next steps 31
In the next chapter 32
3 Understanding the Basic Structure of a Project 33
Basics of Kotlin cross-platform project structure 37

Targets 37
AGP 9 and the new Android-KMP plugin (com.android.kotlin.multiplatform.library) 39
Source Sets 42
Advanced Concepts of Cross-Platform Project Structure 46
Example: Cross-platform dependencies with Ktor 49
Dependency resolution 50
Sharing code across platforms 51
Summary and lessons learned 57
Practical skills acquired 60
Checkpoints 60
In the next chapter 61
4 First Project in Kotlin Multiplatform 62
Introduction 62
Prerequisites 63
Step 1: Prepare the development environment 65
Step 2: Create a new KMP project 66
Step 3: Understand the project structure 68
Step 4: Add shared logic with expect/actual 71
Step 5: Build the Android app 74
Step 6: Build the iOS app 77
Step 7: Run and test on both platforms 79
Step 8: Troubleshoot common issues 79
Step 9: Apply minimum best practices (so the project can scale) 81
Verification checklist 83
Closing 83
In the next chapter 83
5 Introduction to Dependency Injection in KMP 84
Framework-free approaches 87
Dependency Injection Frameworks 91
Advanced section: Metro (compile-time DI with compiler plugin) 101
Comparison and Selection Criteria “Automated” 103
Key points to master 105

In the next chapter 106
6 Modularization 107
Why modularize in Kotlin Multiplatform 108
When NOT to modularize 109
Modularization strategies (and how to choose) 109
Basic modularization: a single shared module 110
Advanced modularization: multiple shared modules 111
The iOS challenge: why multiple frameworks can fail 112
Why use an “umbrella framework” in iOS 113
Repository configurations (according to JetBrains documentation) 120
Note on Swift Export (experimental) 121
Considerations about compilation times 121
Summary 122
In the next chapter 122
7 Testing 123
Why testing is crucial in KMP 124
Select testing tools for KMP 125
Use the testing pyramid as a guide in KMP 125
Practical example: Rick and Morty application 127
Run your tests (IDE and Gradle) 132
Isolate slow dependencies with test doubles 136
Implement unit tests 136
Integration tests (Ktor and SQLDelight) 144
Measure coverage in KMP with Kover (and understand its limitations) 154
Best practices for testing in KMP 156
Specific rules for cross-platform testing 157
Summary and lessons learned 157
Verification checklist 158
In the next chapter 158
8 Integrating Native Dependencies into KMP 159
When to integrate native libraries? 160
Balance of advantages and disadvantages 163
Adding Android dependencies to KMP 164

Adding iOS dependencies to KMP 165
Summary and lessons learned 185
In the next chapter 186
9 Designing KMP libraries with native APIs for Android and iOS 187
Introduction: the real challenge is not “code sharing” 188
Part 1: Design APIs that respect each platform (without duplicating logic) 189
Part 2: Concurrency: expose asynchrony without penalizing iOS 203
Part 3: Design types and contracts that feel native in Swift 210
Part 4: Distribution and production considerations 214
Summary: what you should take away from this chapter 221
Verification checklist 221
In the next chapter 222
10 Swift Export 223
Why Swift Export exists 224
The problem: limitations of classic interoperability with Objective-C 224
What Swift Export is (and isn’t) 226
How Swift Export works: architectural overview 226
Complex types: current limitations (summary) 232
Configuring Swift Export in your project 233
The future of Swift Export (orientation) 239
Conclusion 240
Verification checklist 240
In the next chapter 241
11 Libraries 242
How to choose libraries without compromising commonMain 244
Networking: choose an HTTP library that scales 246
Storage: preferences, key-value, and simple objects 250
Database: structured persistence and queries 256
Jetpack Multiplatform: where to look and how to validate 259

Where to find more libraries (and how to evaluate them) 259
Suggested process for choosing libraries (reproducible) 260
Summary: criteria that matter most in KMP 260
Verification checklist 260
In the next chapter 261
12 Essential Tools and Plugins for Cross-Platform Development... 262
Diagnose your environment before blaming the project 263
Improves Kotlin ⇄ Swift interoperability when iOS DX degrades 264
Distribute iOS binaries in a repeatable manner (SwiftPM / CocoaPods) 266
Accelerate the development experience in the IDE and project startup 268
Generate verifiable documentation for your public API 269
Maintain quality and consistency in shared code 269
Evaluate build tools with discretion (no promises) 271
Amper 271
Quick guide: if X happens, look at Y 272
Summary and lessons learned 272
Verification checklist (for this chapter) 273
References 274
About the Author 284

Preface

This book is aimed at **Mobile Developers** with experience in Android (Kotlin) or iOS (Swift) who want to adopt **Kotlin Multiplatform (KMP)** while maintaining rigorous architecture standards and sustainable practices. It is also useful for **architects, technical leaders, and teams** evaluating or already implementing code-sharing strategies.

Target Audience

- Professionals with experience in **Android** or **iOS** looking to adopt KMP with a solid foundation.
- Teams that need to **share business logic** across platforms while maintaining native UIs.
- Professionals who require a comprehensive guide on **modularization**, **DI**, **testing**, and **native integration**.

What You'll Learn

This book guides you from strategic decision-making to practical execution: structure, architecture, integration, and distribution.

- **Make informed decisions**: evaluate whether KMP is suitable for your use case and understand the cross-platform ecosystem.
- **Adopt KMP**: benefits, limitations, adoption strategies, and how to present it to your team.
- **Master project structure**: targets, source sets, AGP 9, and shared code organization.

- **Build a real project**: connect theory with scalable decisions.
- **Design maintainable architecture**: DI (including Metro), modularization, and testing.
- **Integrate with the native world**: native dependencies, designing natural APIs for Android and iOS, concurrency, and distribution.
- **Expose code to Swift**: Swift Export, libraries, and essential tools.

What This Book Doesn't Cover

- This is not a cross-platform UI manual. The focus is on **shared logic and architecture**.
- It doesn't cover "every library." Instead, it teaches **criteria** for choosing and integrating.

How to Use This Book

- If you're new to KMP, read it **in order**.
- If you already have experience, go straight to **DI**, **Modularization**, and **Testing**.
- If you design SDKs, prioritize **library design** and **native integration**.

Acknowledgments

This book is not built in solitude. Thanks to those who contributed ideas, reviews, questions, and feedback that helped improve its clarity and practical focus.

A special thanks to the **Kotlin and Kotlin Multiplatform community**, whose libraries, examples, and technical discussions accelerate learning and raise the quality standard of the entire ecosystem.

1

What is cross-platform mobile development?

Modern product teams often build and maintain applications for Android and iOS in parallel. Doing so with two separate codebases increases costs, lengthens *time-to-market*, and increases the risk of functional inconsistencies between platforms.

Cross-platform mobile development emerges as a strategy to **share part of the code** between platforms and reduce that friction, without necessarily sacrificing a native experience.

Android and iOS account for the vast majority of the mobile market, so evaluating a sharing strategy is often relevant for many products. Still, it's not a universal requirement: it depends on your context, your team, and the type of application you're building.

This chapter introduces you to the main approaches, when it makes sense to choose each one, and how Kotlin Multiplatform (KMP) fits into that landscape.

What you will learn

- What "cross-platform development" means in practical terms (what is shared and what is not).
- What are the most common approaches to building Android and iOS

applications, and their typical trade-offs.
- A practical criterion for deciding between native and cross-platform based on product and team variables.
- Where Kotlin Multiplatform (KMP) stands compared to other popular alternatives.

Prerequisites

- Previous experience developing for Android or iOS (SDK, tooling, and build cycle).
- Basic familiarity with architectural concepts (layers, data models, business logic).
- You do not need previous experience with KMP to follow this chapter.

How to read this chapter

- If you are deciding on a strategy, focus on "Choosing an approach..." and "Making a practical decision...".
- If you have already decided to explore KMP, focus on "Kotlin Multiplatform (KMP)" and the "Summary."

Understanding what "multiplatform" means in practice

Cross-platform mobile development allows you to create applications that run on multiple mobile operating systems **by sharing a common code base**. This sharing can span different layers of the architecture:

- Utilities and domain logic.
- Data models and serialization.
- Networking and data access.
- In some approaches, also the user interface.

In practice, the idea is **to avoid rewriting** when the functionality is conceptu-

ally the same on Android and iOS, while retaining the option to integrate native code when needed (for example, for system-specific APIs or very specific UI requirements).

Choosing an approach for Android and iOS (and understanding its trade-offs)

Broadly speaking, there are four common approaches. None is "the best" in the abstract: each optimizes a different set of variables such as performance, user experience, development speed, and maintainability.

1) Develop native apps per platform (two code bases)

What it is: a specific application for each operating system, built with its native SDK and toolchain.

- Android Studio + Kotlin/Java for Android.
- Xcode + Swift/Objective-C for iOS.

What you share: Typically, you don't share production code (at most, you align logic through specifications, tests, or equivalent libraries per platform).

Typical advantages:

- Direct and complete access to system APIs.
- Predictable performance.
- UX aligned with native patterns without friction.

Typical limitations:

- Duplicate effort for equivalent logic.
- Higher coordination and simultaneous delivery costs.

Use cases where it usually fits well: applications with very high performance requirements (e.g., 3D gaming), deep integration with platform-specific capabilities, or teams with strong platform specialization and the ability to maintain two code bases over the long term.

2) Use Progressive Web Apps (PWA)

What it is: applications built with web technologies (HTML, CSS, JavaScript, WebAssembly) that run in the browser and can behave like installable apps.

What you share: virtually the entire application, because the execution platform is the browser.

Typical advantages:

- Simple distribution.
- More direct publication cycle.
- High code reuse.

Typical limitations (vary by platform and version):

- Limited access to device capabilities compared to native SDKs.
- Performance and UX conditioned by the browser and integration with the operating system.

> **Note:** On iOS, support for some capabilities (e.g., notifications or hardware integrations) has changed over time and may vary by version. Validate specific limitations with official documentation before making a decision.

3) Build cross-platform apps (sharing modules between Android and iOS)

What it is: a single project shares a substantial part of the code and is delivered on multiple platforms. Depending on the framework, you can share only the logic or also the UI.

What you share: typically, business logic, models, networking, and persistence. In some cases, also the UI.

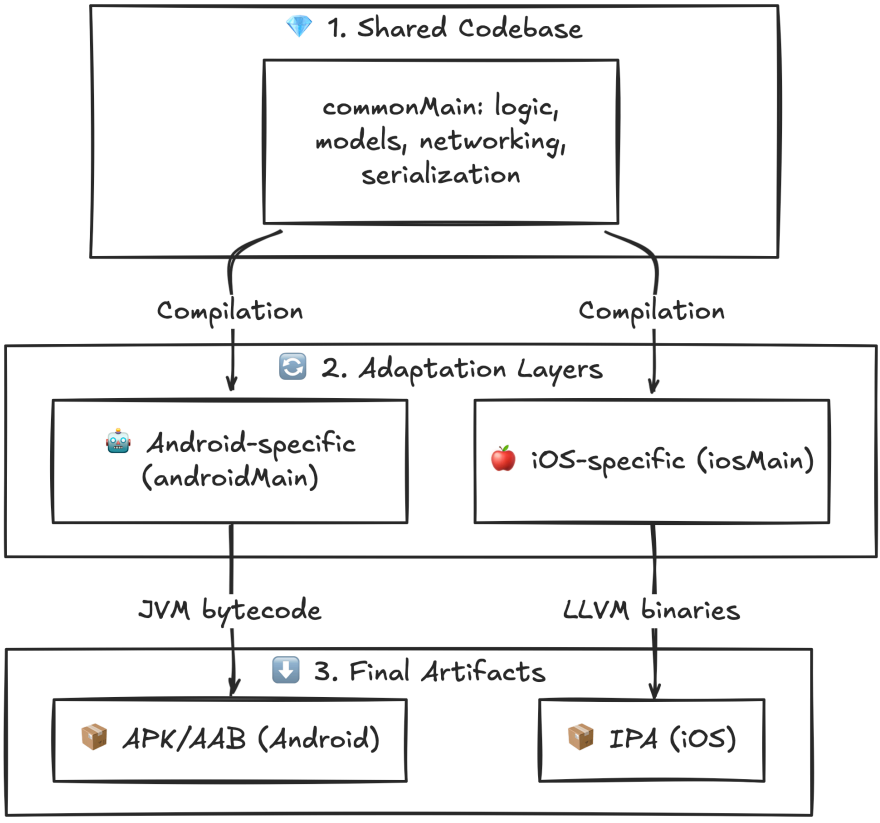
Typical advantages:

- Less duplication.
- Greater functional consistency across platforms.
- Faster iteration on shared features.

Typical limitations:

- Integration with native APIs when you need specific capabilities.
- Deciding what to share (and what not to share) is a key skill for avoiding friction.

Diagram (conceptual): which part is shared and which part is platform-specific.



4) Develop hybrid apps (Web + WebView + native bridge)

What it is: applications built with web technologies (HTML/CSS/JavaScript) that run within a native WebView (e.g., WKWebView on iOS or WebView on Android), with native plugins acting as a bridge to access system APIs.

What you share: much of the UI and web logic.

Typical advantages:

- High code reuse.

- Rapid iteration in web layers.

Typical limitations:

- Performance and UX conditioned by WebView.
- Fine integration with native components is often more complex than in native apps.

Making a practical decision: native vs. code sharing

The discussion is not binary. It is useful to think in terms of a **spectrum**: you can share from "almost nothing" to "almost everything," and that scope can change over time.

In general:

- If your priority is **maximum integration with system APIs** or **maximum graphics performance**, the native approach usually has clear advantages in those areas.
- If your priority is **functional consistency**, **speed of delivery**, and **reduction of duplication**, code sharing often adds value, especially at the domain and data layer.
- Many companies choose a middle ground: **shared logic with native UI**, and evaluate whether it is appropriate to expand sharing in stages.

> "*The need for fast delivery led us to experiment with a multi-platform architecture. Now we are taking it a step further by using Kotlin Multiplatform to write the business logic once in Kotlin and compile it into a library for Android and a native framework for iOS.*" — Netflix Studio Apps Team (2023)

Assess whether the multiplatform approach is right for you

The answer depends on business goals, UX expectations, and team capabilities. These factors often tip the balance.

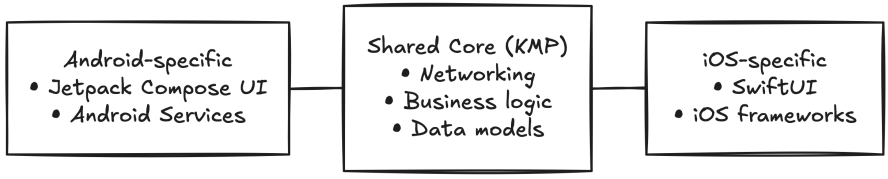
Common benefits (and what they typically enable)

Technical benefits:

- Reduce duplication in serialization, networking, validations, and business rules.
- Avoid functional differences between Android and iOS.
- Accelerate changes when the logic is the same on both platforms.

Team and maintenance benefits:

- Share knowledge between Android/iOS profiles in the common layer.
- Simplify maintenance by reducing "two equivalent implementations."
- Improve release consistency when there are features that must be released simultaneously.



Typical challenges (and how to mitigate them)

Legitimate concerns about performance and access to native capabilities persist. The outcome depends, above all, on two decisions:

- **What you share:** sharing domain and data is usually more stable than sharing UI (although it depends on the framework and the product).
- **How you integrate with native:** even with shared code, you will need integration points with platform-specific APIs.

In the case of KMP, shared code is compiled to bytecode or native binaries depending on the target and can be integrated with platform APIs using theexpect/actual pattern, avoiding layers of interpretation or bridges typical of shared runtime approaches.

Specific example (conceptual): in KMP, commonMain is compiled to JVM bytecode for Android and to native binaries using Kotlin/Native (LLVM backend) for iOS.

Understanding the landscape: Flutter, React Native, and Kotlin Multiplatform

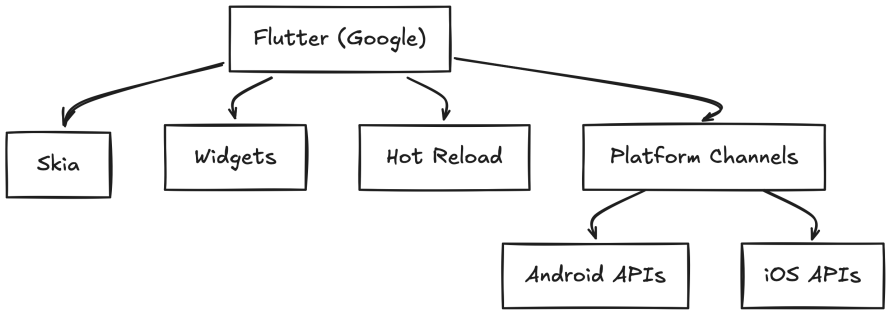
The most commonly used options include Flutter, React Native, and Kotlin Multiplatform. Each offers a different model of sharing, tooling, and execution.

Flutter

Google's Dart-based framework that renders its own UI using the Skia graphics engine, without relying on native operating system components. It stands out for its Hot Reload and a robust ecosystem of widgets.

It is usually a good fit when you want to share the entire UI, iterate quickly (e.g., prototypes and MVPs), or control the design consistently across

platforms. Platform Channels are used for specific capabilities.



React Native

Meta framework that uses JavaScript and React to describe interfaces that translate into native views. Historically, it used a "bridge" between JavaScript and native modules, which can have an impact in intensive scenarios.

The architecture has evolved (e.g., with JSI in the "New Architecture") to improve performance and communication with native. Even so, the typical fit depends heavily on the team profile and the type of application.

Kotlin Multiplatform (KMP)

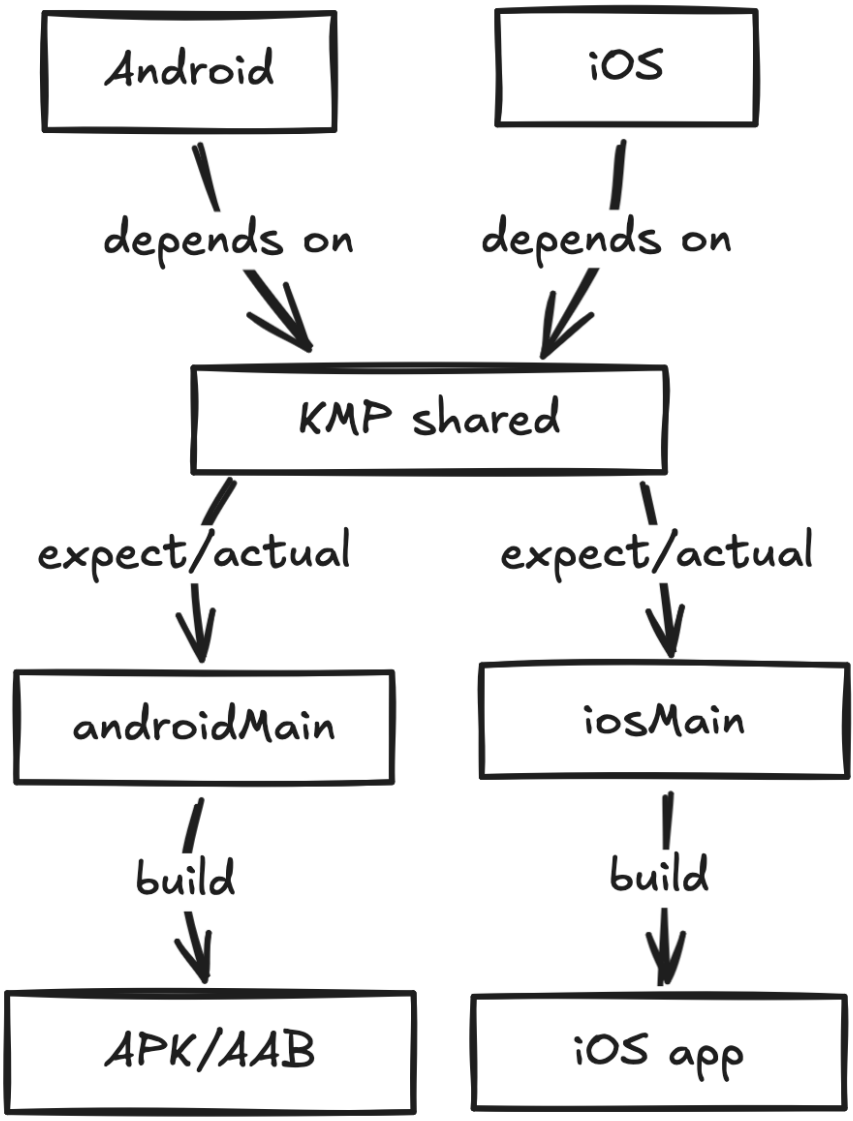
JetBrains technology for sharing code between Android, iOS, desktop, web, and server while maintaining the advantages of native development. Unlike frameworks that completely abstract the platform, KMP takes a **native-first** approach: shared code is integrated as just another dependency in existing Android and iOS projects, without imposing a specific architecture.

Key points:

- **Flexible sharing:** from utilities to business logic and, if needed, UI with

Compose Multiplatform.

- **Direct native integration:** interoperability with platform APIs and the ability to implement specific details usingexpect/actual.
- **Incremental adoption:** you can introduce shared modules into existing projects and scale gradually.
- **Continuity with Kotlin:** if you already use Kotlin on Android, you reduce risk by reusing language, tooling, and practices.



Companies such as McDonald's, Netflix, 9GAG, VMware, Cash App, and Philips report improvements by sharing critical logic, accelerating launches, and maintaining native UIs where it most impacts the experience.

Summary

Cross-platform development does not replace native development; it complements it. Choosing wisely what to share and what to keep native allows you to balance speed, cost, and user experience.

Flutter, React Native, and KMP are valid options with different profiles. Throughout this book, you will learn how to use Kotlin Multiplatform to maximize code reuse without sacrificing the native experience, integrating shared modules with existing Android and iOS projects. You will work with *source sets*, theexpect/actual mechanism, advanced Gradle configuration, and essential ecosystem libraries such as Ktor (networking), SQLDelight (persistence), Kotlinx.serialization, and Compose Multiplatform (optional shared UI).

By the end, you will have criteria for deciding the scope of sharing in your application, a practical guide for introducing KMP to your team, and solid patterns for delivering functionality simultaneously on Android and iOS with a single shared logic base.

Verification checklist

- I can explain what "multiplatform" means and which layers of an app can be shared.
- I can describe the four approaches (native, PWA, multiplatform, hybrid) and a typical trade-off for each.
- I can list at least 3 variables to decide (UX, performance, *time-to-market*, team capabilities).
- I can explain why many organizations choose "shared logic + native UI" as a starting point.
- I can describe KMP's *native-first* approach in one sentence.

In the next chapter

We will clarify why Kotlin Multiplatform is a strong option for sharing logic without sacrificing a native experience. You will see what is worth sharing, what is not, and how to plan a gradual adoption for a real project.