Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

Adopting Kotlin Multiplatform in the Future Lab Aachen App

Bachelor's Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Lennart Fischer

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Ulrik Schroeder

Registration date: 19.08.2024
Submission date: 19.12.2024

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Declaration of Academic Integrity

Fischer, Lennart	433990
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

Adopting Kotlin Multiplatform in the Future Lab Aachen App

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting. I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Aachen, 15.12.2024	L. Fischer
Ort, Datum/City, Date	Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 15.12.2024	L. Fischer
Ort, Datum/City, Date	Unterschrift/Signature

Contents

List of Figures and Tables

Abstract

This thesis presents the adoption of Kotlin Multiplatform (KMP) to facilitate a cross-platform architecture for the Future Lab Aachen App (FLApp), an interactive mobile guide to Aachen's scientific and historical landmarks. Due to being built as two native implementations for iOS and Android, FLApp suffered from code duplication, platform inconsistencies, and maintainability issues. These problems motivated the migration to a unified, cross-platform architecture while maintaining feature parity and enhancing developer productivity.

In our work, we evaluated various cross-platform framework options and selected KMP for FLApp because of the native-oriented features and incremental adoption capabilities. We executed a migration plan that involved transitioning from Objective-C to Swift and incrementally adopting KMP to unify the business layer. Later, we evaluated the achievement of our goals and our developer experience using KMP for FLApp.

We mitigated the previous issues of FLApp and created a stable foundation for future work. We were able to significantly improve maintainability, code reuse, and architectural consistency while preserving native user interfaces. The results demonstrate that KMP is a suitable framework for native-to-cross-platform transitions. The insights gained from this project provide valuable guidance for similar migration efforts, covering everything from framework selection to execution.

Überblick

Diese Arbeit untersucht die Einführung des Cross-Platform Frameworks Kotlin Multiplatform (KMP) in der Future Lab Aachen App (FLApp). Die FLApp ist ein interaktiver Guide, der die wissenschaftlichen und historischen Sehenswürdigkeiten Aachens präsentiert. Ursprünglich wurde die FLApp als native iOS- und Android App implementiert. Allerdings traten mit der Zeit erhebliche technische Herausforderungen wie inkonsistentes Verhalten zwischen Plattformen aufgrund von dupliziertem Code und Wartbarkeitsprobleme auf.

Um diese Probleme zu beheben, strebten wir die Migration zu einer einheitlichen, plattformübergreifenden Lösung an, die zwar die Produktivität in der Entwicklung steigern, aber die aktuelle Funktionalität nicht beeinträchtigen sollte. Dafür führten wir eine detaillierte Evaluation der unterschiedlichen Framework-Optionen PWA, React Native, Flutter und KMP für die FLApp durch, die uns schlussendlich wegen der guten Interoperabilität mit den beiden Plattformen zu KMP geführt hat. Der Migrationsprozess umfasste mehrere Schritte, darunter die Migration von Objective-C auf Swift in der iOS App, die einheitliche Implementierung der Geschäftslogik und Datenverwaltung in KMP sowie die Umsetzung von Lösungen zur Verbesserung der Codequalität und Konsistenz.

Die Ergebnisse zeigten, dass KMP ein geeignetes Framework für die Cross-Plattform Migrationen von nativen Apps ist. Es ermöglicht erhebliche Verbesserungen in der Wartbarkeit, Codewiederverwendung und strukturellen Konsistenz, während beispielsweise die native Benutzeroberfläche beibehalten werden kann. Diese Migration löste nicht nur bestehende Probleme, sondern brachte auch eine gute Grundlage für zukünftige Wartung und neue Funktionen. Von der Auswahl eines geeigneten Frameworks bis hin zur Umsetzung, können unsere Erkenntnisse wertvoll für ähnliche Migrationsprojekte sein.

Acknowledgments

First, I would like to thank my supervisors, Kevin Fiedler and Oliver Nowak, for their helpful support and advice while working on this thesis.

I also want to thank Prof. Dr. Jan Borchers and Prof. Dr. Ulrik Schroeder for their time and expertise in examining this thesis.

Lastly, I want to thank my family and friends, who have always supported me and my work.

Conventions

Throughout this thesis we use the following conventions:

- The thesis is written in American English.
- The first person is written in plural form.
- For unidentified third persons we use the pronoun they/their.

Short excursuses are set off in colored boxes.

EXCURSUS:
Excursuses are set off in orange boxes.

Where appropriate, paragraphs are summarized by one or two sentences that are positioned at the margin of the page.

This is a summary of a paragraph.

Source code and implementation symbols are written in `typewriter-style text`.

Chapter 1

Introduction

The Future Lab Aachen App[1] (FLApp) is an interactive mobile guide to Aachen's scientific and historical sites. It uses GPS navigation and augmented reality to present multimedia content like audio, videos, and 3D models. Initially developed in 2017, the app runs natively on iOS and Android. It was initially built with Objective-C for iOS and Java, later switching to Kotlin for Android.

FLApp is a native app available on iOS and Android.

While native apps provide vendor-optimized performance and deep integration into the platform, building and maintaining two native apps can be time-consuming and require knowledge of both technologies. Code duplication may introduce inconsistencies between the platforms, and new features and bug fixes must be addressed individually [Amatya and Kurti, 2014; Corral et al., 2012].

Building and maintaining two native apps requires significant effort.

Having separate native implementations for iOS and Android brought several challenges to FLApp's development. One major issue arose in how each platform handled content parsing from the local `plist` files. This issue caused inconsistencies in the display of content on both platforms. Moreover, the architectural approaches diverged between platforms: iOS utilized the Model-View-Controller (MVC) pattern, while Android implemented the Model-View-View Model (MVVM).

FLApp also faced problems due to individual native implementations.

[1] https://futurelab-aachen.de/en/app

Overview of cross-platform development approaches.

Cross-platform mobile frameworks offer solutions for improving maintainability, removing code duplication, and avoiding inconsistencies in the app experience [Corral et al., 2012; Amatya and Kurti, 2014]. Developers can build applications using web-based technology like Progressive Web Apps (PWAs) or Ionic. Frameworks such as React Native or Flutter offer another approach to cross-platform development. Kotlin Multiplatform (KMP) offers a more native-oriented solution and enables the sharing of business logic while keeping native UI implementations and the flexibility to integrate platform-specific frameworks as needed.

Migration goals: UX, shared code, and improve maintainability.

We aimed to migrate FLApp to cross-platform technology while ensuring a good user experience (UX), establishing a unified codebase with consistent architecture, and enhancing overall maintainability.

We selected KMP as the best fit for FLApp.

Based on our cross-platform technologies and literature review evaluation, KMP emerged as a suitable solution for FLApp. KMP enables us to maintain the native user interface and complex features like Augmented Reality and iBeacon Tracking on iOS and Android while easily sharing the underlying data model layers between both app versions. This technology selection ensures a consistent app architecture across both platforms.

KMP brought good developer experience, but has some limitations in interoperability.

KMP provides a good developer experience, although Kotlin-Swift interoperability currently relies on Objective-C, complicating integration with Swift-only frameworks. JetBrains' upcoming direct Kotlin-to-Swift export[2] feature promises to improve up on this issue.

KMP was a great choice for FLApp, and we achieved our migration goals.

Selecting KMP for FLApp's cross-platform transition proved to be a good decision. We maintained our native user interface and avoided a full technology migration. Implementing a shared data layer allowed us to reduce code duplication, resolve existing bugs, and ensure consistent content parsing across platforms without introducing regressions or performance issues. Our new app architecture will empower new developers to streamline future up-

[2] https://blog.jetbrains.com/kotlin/2024/10/kotlin-multiplatform-development-roadmap-for-2025/#kotlin-to-swift-export

dates and simplify maintenance by consolidating core logic across iOS and Android.

In Chapter 2, we present cross-platform approaches, respective frameworks, and examples for native-to-cross-platform migrations. Chapter 3 describes the goals of our migration and the decision process, along with the most important criteria and why we finally selected KMP for FLApp. We describe the migration process of FLApp to KMP in Chapter 4. Then, we evaluate the satisfaction of the requirements and our experience, especially from a development perspective, in Chapter 5. Finally, in Chapter 6, we summarize our work and propose future work for cross-platform research and regarding FLApp.

Chapter 2

Related Work

2.1 App Development Approaches and Technologies

Since the first mobile devices were introduced, many approaches have been taken to developing apps for the mobile operating systems iOS and Android. One of the fundamental distinctions in mobile app development lies between native and cross-platform approaches [Nawrocki et al., 2021].

Native vs. cross-platform is a key distinction in mobile development.

The term *native app* was first popularized by Steve Jobs[1] when he introduced the first native APIs and tools for developing iOS apps. Native apps are developed using the first-party software development kits (SDKs) offered by platform vendors like Apple and Google. These SDKs use platform-specific programming languages such as Objective-C and Swift for iOS and Java or Kotlin for Android [Nunkesser, 2018; Amatya and Kurti, 2014].

Native apps use first-party SDKs provided by platform vendors.

In contrast, cross-platform development (CPD) refers to the practice of creating software for multiple platforms from a single codebase. As stated by Segun-Falade et al. [2024], this eliminates the use of two individual code bases, en-

Using cross-platform development creates apps for multiple platforms using a single codebase.

[1] https://itunes.apple.com/de/podcast/wwdc-2008-keynote-address/id275834665

abling developers to write code once and deploy it across different operating systems with minimal adjustments.

Cross-platform frameworks provide tools for building cross-platform apps.

Cross-platform frameworks (CPFs) consist of tools, libraries, and plugins that enable cross-platform development. These frameworks offer a unified development environment, components for user interfaces (UI), and other essential functionality. Often, additional platform-specific code allows performance improvements or the implementation of capabilities that the CPF itself does not offer.

2.1.1 Taxonomy of App Technologies

Taxonomies of app technologies help group various technological approaches.

There are many different technological approaches to how cross-platform frameworks achieve CPD. Therefore, early on efforts were made to classify the CPFs into different categories depending on the technological approach.

Web/Hybrid/Native taxonomy is widely used but not very expressive.

Till today, the terms Web, Hybrid, and Native (WHN) are often used to classify mobile development approaches [Behrens, 2010; Pinto and Coutinho, 2018]. This categorization became increasingly vague as more and more frameworks fell into the hybrid category. This ambiguity has driven the development of more nuanced classifications to describe modern app technologies [Nunkesser, 2018].

Behrens was amongst the first to extend the WHN classification.

Early extensions to the WHN taxonomy by Behrens [2010] added two categories: Interpreted Apps, which combine native UI elements with platform-independent logic, and Generated Apps, which produce fully native applications for each platform from a single code base. Rahul Raj and Tolety [2012] categorized approaches as Web, Hybrid, Interpreted, and Cross-compiled, while El-Kassas et al. [2017] proposed a more detailed classification, including Compilation, Component-based, Interpretation, Modeling, Cloud-based, and Merged approaches.

1. **Endemic Apps** use the native SDKs provided by the OS vendors. These apps offer vendor-optimized performance and tight integration with the platform.
2. **Pandemic Apps** group cross-platform approaches supported natively by all major OSs. This includes:
   - **Web Apps** leverage HTML, CSS, and JavaScript to create apps accessible through a browser or a minimal app shell.
   - **Hybrid Web Apps** use frameworks like Ionic to embed web technologies within a native app container.
   - **Hybrid Bridged Apps** bridge JavaScript with native UI elements through frameworks such as React Native.
   - **System Language Apps** leverage shared system-level languages like C++ for game engines (e.g., Unity, Unreal Engine) or frameworks like Qt.
3. **Ecdemic Apps** are developed using a language or toolset not native to the OS. For example:
   - **Foreign Language Apps** are built using frameworks like Flutter or Kotlin Multiplatform that compile non-native languages into native code for each platform.

Figure 2.1: Categories and subcategories proposed by Nunkesser [2018] to classify different mobile development approaches.

Building on earlier classifications, Nunkesser [2018] introduced a taxonomy that organizes mobile development approaches into the three main categories Endemic Apps, Pandemic Apps, and Ecdemic Apps along with subcategories for a more detailed comparison (Figure 2.1).

Modern CPD approaches can be categorized as Endemic, Pandemic, and Ecdemic.

This taxonomy, visualized in Figure 2.2, offers a clear and structured overview of the foundational approaches to mobile cross-platform development. For us, it serves as a helpful framework for selecting representative technologies to explore in detail.

Nunkesser's taxonomy provides a clear foundation for exploring cross-platform approaches.

Using this taxonomy, we researched popular CPF representatives based on literature and usage statistics for each category. We decided to exclude Hybrid Web Apps from

We selected popular representatives of the taxonomy's categories for detailed research.

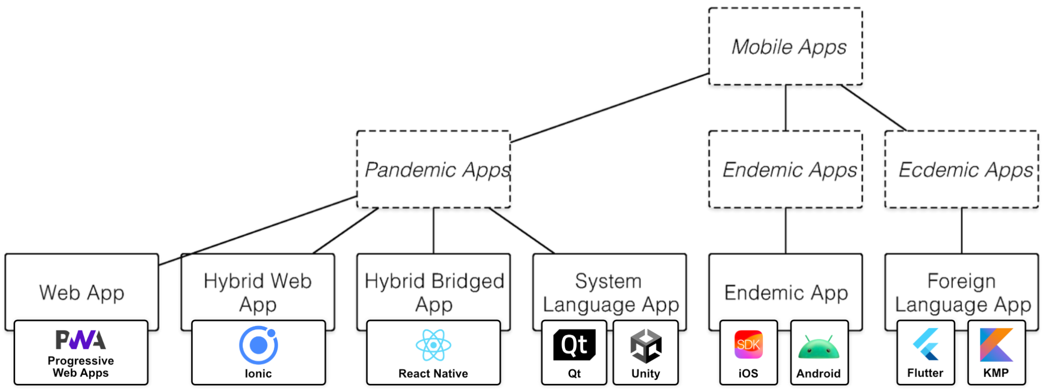


Figure 2.2: Categorization of cross-platform technologies proposed by Nunkesser [2018] adapted to include representative framework.

this selection due to their significant overlap with Hybrid Bridged Apps, which we prioritize due to their enhanced capabilities despite both using JavaScript primarily. Similarly, System Language Apps are excluded due to limited available literature[2], particularly regarding Qt[3]. We also will not present Endemic (Native) Apps as the objective of our work remains the cross-platform migration.

A detailed evaluation is provided in chapter 3, aligned with FLApp's migration needs.

A comparative review of cross-platform frameworks is deferred to chapter 3, where we evaluate these technologies in the context of FLApp's migration requirements. This approach ensures that decisions are informed by practical needs rather than abstract theoretical considerations.

2.1.2 Progressive Web Apps

PWAs use web technologies to provide app-like experiences.

Progressive Web Apps (PWAs)[4] leverage web technologies (HTML, CSS, and JavaScript) to provide an app-like experience across multiple platforms. After being developed as

[2] This does not hold for Game Engines, but they do not play a role in our research.
[3] https://doc.qt.io/qt-6/mobiledevelopment.html
[4] https://web.dev/progressive-web-apps

an open web standard, PWAs have been adopted by major companies, including Twitter[5], Uber[6], and Pinterest[7].

Enhanced capabilities can be implemented using service workers and web app manifests. Service Workers[8] enable offline caching, background synchronization, and push notifications.

Services workers enable offline caching and background synchronization.

While PWAs can be used in a web browser like any regular website, they can also be added to the home screen, providing an app-like experience. Through tools like PWABuilder[9], PWAs can be bundled into applications shipped to mobile app marketplaces.

PWAs can be distributed via a web browser but also through app marketplaces.

PWAs can use Web APIs[10] for advanced platform integration with the operating system (OS). These APIs include geolocation, camera access, basic sensor data, and simplified push notifications. However, compared to other CPFs, access to advanced device features remains limited, with browser vendors and platform providers maintaining strict control over API exposures.

PWAs can access some advanced device capabilities via Web APIs.

Developers can create PWAs using popular web frameworks like React, Angular, and Vue.js. Modern browsers provide debugging and performance analysis tools[11] specifically designed for PWA development. Frameworks like Workbox[12] simplify service worker implementation and caching strategies.

Browser developer tools help to build and improve PWAs.

[5] https://web.dev/case-studies/twitter
[6] https://www.uber.com/en-EC/blog/m-uber
[7] https://business.adobe.com/blog/basics/progressive-web-app-examples
[8] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
[9] https://www.pwabuilder.com
[10] https://developer.mozilla.org/en-US/docs/Web/API
[11] https://developer.chrome.com/docs/devtools/progressive-web-apps
[12] https://developers.google.com/web/tools/workbox

2.1.3 React Native

React Native is widely used across the app marketplaces.

React Native[13] is an open-source, cross-platform framework for developing mobile applications for iOS and Android. React Native was initially developed by Facebook in 2015 after they experienced problems using HTML5 for the mobile version of Facebook[14]. React Native is used by many large companies like Microsoft, Amazon, and Shopify[15]. According to *App Figures* SDK analysis[16] (November 2024), 13% of the free iOS apps and 16% of the free Android apps are built using React Native.

A JavaScript bridge controls native user interface.

Unlike other frameworks (cf. Hybrid Web Apps) that rely on webviews to render the user interface, React Native directly utilizes the host platform's native rendering APIs. React Native utilizes a "bridge" that translates JavaScript calls into the corresponding native APIs, allowing seamless communication between the JavaScript code and the native platform features [Eisenman, 2015].

React Native uses JSX for user interfaces and exposes several platform APIs.

User Interfaces are written in JSX[17], a syntactical combination of HTML and JavaScript. These components describe the structure and appearance of the user interface. A simplified subset of CSS is used for styling, primarily relying on Flexbox for layout. React Native also provides access to various host platform APIs, including device features like camera, location, and storage, made available through asynchronous JavaScript interfaces [Zammetti, 2018].

Native Modules and Native Components allow interoperability with native code.

Native modules[18] written in platform languages expose their functionality to the JavaScript code through the bridge, allowing JavaScript code to invoke native methods and receive data back. *Native Components* provide a way to wrap native UI elements and expose them as React components that can be used within the JavaScript codebase [Paul and Nalwaya, 2019].

13 https://reactnative.dev
14 https://mashable.com/archive/html5-biggest-mistake
15 https://reactnative.dev/showcase
16 https://appfigures.com/top-sdks/development/all
17 https://facebook.github.io/jsx
18 https://reactnative.dev/docs/native-platform

There is no dedicated Integrated Development Environment (IDE) for writing React Native apps. However, plugins can be used in IDEs like IntelliJ[19] or Visual Studio Code to improve syntax highlighting, code completion, debugging, and on-device testing. Hot reloading allows developers to get feedback on changes in code without restarting the app. Additionally, React Native DevTools[20] and Chrome Developer Tools can be used to debug JavaScript code.

React Native has IDE plugins and additional developer tools.

2.1.4 Flutter

Flutter[21] is an open-source framework for developing cross-platform mobile, web, and desktop applications from a single codebase. Google originally developed Flutter in 2017, and it is used by prominent organizations like Google, Alibaba, BMW, and eBay[22]. According to *App Figures* SDK analysis[23] (November 2024), 12% of the free iOS apps and 18% of the free Android apps are built using Flutter.

Flutter is a widely used cross-platform framework by Google.

Flutter utilizes the Dart programming language to produce machine code via just-in-time (JIT) and ahead-of-time (AOT) compilation of the Dart VM[24]. The framework renders the application using the Skia and Impeller[25] graphics engines. This approach ensures consistent rendering across platforms by drawing every pixel directly, which provides exact UI reproduction regardless of the target device.

Flutter uses a Dart as the primary language.

Developers compose user interfaces using Dart and Flutter's declarative widget-based approach. These components describe the entire visual hierarchy through composable UI elements called widgets. Styling and layout use the composition of widgets using grids, columns, rows, and

Widgets describe the user interface hierarchy.

19 https://www.jetbrains.com/help/idea/react-native.html
20 https://reactnative.dev/docs/react-native-devtools
21 https://flutter.dev
22 https://flutter.dev/showcase
23 https://appfigures.com/top-sdks/development/all
24 https://dart.dev/overview#platform
25 https://docs.flutter.dev/perf/impeller

stacks [Payne, 2019]. Cupertino and Material components[26] design systems bring native-like styling of UI elements.

Platform Channels bring interoperability with native code.

Platform Channels[27] allow bidirectional communication between Dart code and native code. This concept allows developers to invoke platform-specific APIs, implement custom native functionality, and integrate existing native modules into Flutter applications [Cheng, 2019].

Flutter plugins are available for IDEs.

Flutter provides a set of development tools, including hot reload for instant code changes, the Flutter CLI, and integration with popular IDEs like IntelliJ IDEA and Visual Studio Code[28]. Flutter DevTools[29] offers debugging features, performance profiling, and widget inspection capabilities [Payne, 2019].

2.1.5 Kotlin Multiplatform (KMP)

KMP enables code sharing across platforms.

Kotlin Multiplatform[30] (KMP) is an open-source framework for cross-platform development. It enables developers to share code seamlessly between iOS and Android while allowing native integration where necessary. Developed by JetBrains in 2017 and stabilized in 2023, KMP has been adopted by prominent companies such as Netflix, Philips, Quizlet, and McDonald's[31].

Kotlin's compiler can compile to JVM and LLVM for iOS.

Kotlin Multiplatform uses a project structure that separates shared logic from platform-specific implementations to achieve code sharing. Kotlin's different compiler backends allow compilation for various platforms. In the case of mobile, it creates JVM bytecode for Android and native LLVM[32] code for iOS.

[26] https://docs.flutter.dev/ui/widgets
[27] https://docs.flutter.dev/platform-integration/platform-channels
[28] https://docs.flutter.dev/tools
[29] https://docs.flutter.dev/tools/devtools
[30] https://jetbrains.com/kotlin-multiplatform
[31] https://jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html
[32] LLVM is the compiler and toolchain technology used on Apple's platforms.

Native Interoperability is a key feature of Kotlin Multiplatform, enabling integration with existing native codebases. Developers can directly call Objective-C, Swift, and Java libraries from Kotlin code and vice versa. Therefore, expect/actual[33] declarations are used to implement this behavior. They allow developers to define shared interfaces with platform-specific implementations (Listing B.1).

KMP allows integration with native libraries for gradual adoption.

The Compose Multiplatform framework[34] optionally extends the declarative UI framework of Jetpack Compose across multiple platforms, enabling developers to share user interface code between Android, iOS, desktop, and web applications using a single Kotlin codebase.

Compose Multiplatform allows UI sharing across platforms using a single Kotlin codebase.

Kotlin Multiplatform integrates with IntelliJ and Android Studio. KMP provides tooling that includes a multiplatform plugin, debugging support, and integration with existing build systems like Gradle. Xcode is used to write iOS-specific implementations. However, JetBrains' new IDE Fleet[35] aims to provide a KMP environment with language support for both Kotlin and Swift, eliminating the need for Xcode for most of the time.

KMP integrates with IntelliJ, Android Studio, and JetBrains Fleet for cross-platform development support.

2.2 Similar Migration Projects

After reviewing potential cross-platform technologies based on the taxonomy outlined by Nunkesser [2018], we aim to assess the suitability of these technologies for the cross-platform migration of FLApp. To gain insights, we examined similar migration projects documented in both scientific literature and practical case studies, with a particular focus on transitions from native implementations to the identified CPF candidates Progressive Web Apps (PWA), React Native, Flutter, and Kotlin Multiplatform (KMP).

[33] This can be compared to the object-oriented interface/implementation paradigm.
[34] https://www.jetbrains.com/compose-multiplatform
[35] https://www.jetbrains.com/fleet

2.2.1 Scientific Research

Zimmermann [2021] researched app migration examples and strategies.

Zimmermann [2021] is one of the few academic contributions focused on migrating native mobile applications to cross-platform solutions. Their work examines general software modernization and migration approaches, cost estimation, and strategies for migrating from native to cross-platform technology.

Two main migration approaches are reimplementation and incremental adoption.

Two migration approaches are compared by Zimmermann [2021]: complete reimplementation and incremental adoption. They note that reimplementation allows for complete modernization but comes with high risks and costs. At the same time, incremental adoption is more flexible but technically challenging due to the interaction between old and new components.

There needs to be more academic research on native-to-cross-platform migrations.

They found a significant gap in academic literature regarding mobile app migrations and a particular shortage of those from native to cross-platform frameworks. Their other sources include blog posts and conference talks from industry sources highlighting the lack of standardized academic research.

The potential CPF migration of Phase6 app illustrates some important factors and questions.

Zimmermann [2021] also discusses the potential migration of the vocabulary app Phase6. They illustrate how factors such as technology selection, feature prioritization, architecture planning, and team alignment play crucial roles in determining success. However, they cautione that these findings are not universally applicable but offer a starting point for further exploration.

Zimmermann calls for more scientific research.

Despite these insights, Zimmermann [2021] concluded that there is no general solution for mobile app migrations. The need for further academic research and industry-standardized methodologies is clear, especially in addressing successful and failed migration attempts, which can offer valuable lessons for future projects.

In another research, Cheon and Chavez [2021] rewrite a native Android classroom quiz app using Flutter. They especially described and compared the architecture of the imperative Android app and the reactive Flutter app. Their biggest challenge was converting the thread-based networking code to asynchronous functions, which resulted in cleaner code. The resulting app is 37% smaller than the Android app – 6949 source lines of code (SLOC) for Java and 4351 SLOC of Dart – and implementing the user interface required most of their time. They only partially recommended migrating native apps to Flutter in 2021 due to the lack of necessary built-in SDK features and tools.

Other research achieved a significant reduction in code using Flutter, but reimplementing the user interface was time-consuming.

2.2.2 Case Studies

As already observed by Zimmermann [2021], the most valuable insights into similar migration projects are found in real-world migration projects. We, therefore, present case studies on native-to-cross-platform migrations for our researched CPF candidates.

Real-world projects offer the most valuable migration insights.

The case studies presented in this thesis were primarily sourced from the official websites or publications of the respective platform providers (e.g., JetBrains, Meta, and Google). It is important to note that these case studies are typically commissioned by the platform providers, either directly or indirectly, and are published or promoted by them.

Case studies are often commissioned and published by framework vendors.

It is important to approach the given examples cautiously, particularly regarding their relevance over time. Cross-platform technologies constantly evolve, so the factors influencing the decision to adopt or reject a particular framework may shift as new advancements and updates emerge.

The case studies must be cautiously approached due to the release date.

Moreover, these case studies focus only on successful and beneficial experiences, often leaving out failed projects or long-term issues that may arise after a decision. This context has to be considered in the analysis to maintain objectivity.

Case studies on migration or development experiences often exhibit a positive bias.

Not all examples are similar to our project in all aspects but share similar goals and priorities.

While the case studies differ in scope, complexity, team size, and commercial focus, they share similar technological needs. These include faster development, easier maintenance, and improved team collaboration. By analyzing their methods, we can adopt practices and tools that support our goals, especially in workflows and integration strategies.

Progressive Web Apps

We found no examples of native-to-PWA migrations.

We could not find any case studies that involved migrating from native apps to PWAs. This is probably the case because PWAs are limited in their functions, and apps could quickly lose capabilities during migration and thus affect the user experience [Majchrzak et al., 2018]. Nevertheless, there are some interesting examples of the use of PWAs.

Twitter Lite achieved near-native performance with a lot less bandwidth usage.

In a case study presented by Google[36] and Twitter[37] in 2007, engineer Nicolas Gallagher explains their experience of implementing the PWA *Twitter Lite*. According to the case study, it achieves near-native app performance while only downloading 600KB of data, compared to the 23.5MB required to download the native Android app. They also noticed a 65% increase in pages per session, a 75% rise in tweets sent, and a 20% drop in bounce rate.

Twitter Lite optimizes for low-coverage users.

The *Twitter Lite* PWA is also optimized for data-sensitive users by serving smaller media files, enabling a data saver mode, and caching resources. These enhancements reduced data consumption by up to 70% in specific scenarios and were especially valuable for users with limited bandwidth and lower-end devices in emerging markets.

[36] https://web.dev/case-studies/twitter

[37] https://blog.x.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite

Other notable case studies on PWA usage include the use at Pinterest[38], Tinder[39] and Uber[40], who found similar advantages in creating lightweight, app-like mobile experiences.

Other companies found similar advantages in PWAs.

React Native

In 2020, Shopify announced their ambitions of migrating all their native applications to cross-platform technology using React Native in their engineering blog[41]. While they had successful native mobile apps, they aimed to get more effective by bringing the power of JavaScript and the web to mobile and consolidating iOS and Android development into a single stack.

Shopify aimed to migrate native apps to React Native.

The acquisition of Tictail, which successfully used React Native at that time, React Native's performance improvements, and the existing use of the similar React framework (web) at Shopify led them to consider a complete switch to React Native. Their early experiments in a few applications showed code sharing of 95%, exceeding the goal of 80%, and the team felt way more productive.

Shopify's early experiments exceeded code sharing goals.

Two years later, in 2022, Mauricio de Meirelles (Shopify Core Mobile team) summarized the progress of Shopify's migration to React Native[42]. The migration effort focused on *Shop*, Shopify's most significant mobile application. They adopted React Native for new features but quickly realized that this would require maintaining three different architectures, and interoperability with existing native code would become very time-consuming.

Shopify initially tried incremental adoption but faced challenges.

Because of this, they then put complete resources into iteratively rebuilding the app in React Native while directly implementing all new features with that technology. There-

Shopify switched to complete iterative rebuild in React Native.

[38] https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154
[39] https://medium.com/@addyosmani/a-tinder-progressive-web-app-performance-case-study-78919d98ece0
[40] https://www.uber.com/en-NL/blog/m-uber
[41] https://shopify.engineering/react-native-future-mobile-shopify
[42] https://shopify.engineering/migrating-our-largest-mobile-app-to-react-native

fore, they introduced an internal React Native training program for native Swift and Kotlin developers. They established migration criteria based on RICE scoring (reach, impact, confidence, and effort) for prioritizing application parts, ensured stable releases of all migrated parts by using feature flags before completely removing the native code, and even introduced an internal dashboard on the migration process. They mention reducing implementation differences between iOS and Android and simpler code as additional benefits of the migration efforts.

Coinbase also migrated to React Native using a complete rewrite.

Coinbase also migrated from native to React Native[43] in 2020 but opted for a complete rewrite due to the lessons learned from AirBnB's failed React Native adoption in 2018[44] where one of the key problems was the incremental adoption of new features, which introduced significant overhead going forward (similar to Shopify's experiences).

Flutter

SNCF operated two separate native apps for 10 years.

Prior to adopting Flutter, french railway operator SNCF ran two separate native applications for ten years[45] (one for long-distance travel and one for day-to-day mobility). The development and maintenance required four separate development teams working with diverse technologies, leading to inconsistencies in development practices, app functionality, and user experience.

SNCF chose Flutter after evaluating KMP and React Native.

SNCF wanted to improve its efficiency and combine its four native implementations into one implementation using a cross-platform framework. They, therefore, built proof-of-concepts for KMP, React Native, and Flutter and had a list of criteria for their selection. They found KMP a good candidate but did not want to write their UIs twice as the Compose Multiplatform framework was unavailable in 2020. React Native did not provide the required developer expe-

[43] https://www.coinbase.com/de/blog/announcing-coinbases-successful-transition-to-react-native
[44] https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a
[45] https://www.youtube.com/watch?v=mKerYStzfIw

rience. SNCF chose Flutter for its good performance, mature state, good developer experience, and trusted partners.

SNCF set a timeframe of 18 months for a complete rewrite of the new app. They chose a Backend-for-Frontend approach, moving all presentation logic, such as model, mapping, and localization, to the server for better consistency and a smaller frontend size. An OpenAPI specification was used to generate API clients in Flutter automatically. They also implemented automated testing, ensuring app stability when iterating quickly.

SNCF executed a full rewrite with a Backend-for-Frontend in 18 months.

By adopting Flutter, SNCF significantly accelerated their time-to-market, enabling weekly feature releases and achieving 90% code sharing across platforms[46]. Although challenges like shader compilation issues on older Android devices emerged, SNCF collaborated with Google to resolve them, with the now-released advancements like Impeller to enhance rendering performance.

Flutter adoption accelerated time-to-market and improved code sharing.

Other notable native to Flutter migrations include BMW's switch from a native iOS app to Flutter, which allowed BMW to release their *My BMW* app for Android for the first time[47].

BMW also migrated from a native iOS app to Flutter.

Kotlin Multiplatform

Cash App transitioned to Kotlin Multiplatform to improve their development speed and collaboration between the iOS and Android teams. Before this migration, the application relied on shared JavaScript code to achieve cross-platform functionality. However, this approach presented challenges in maintaining performance and platform-specific features[48].

Cash App transitioned to KMP to replace shared JS code.

The team first started integrating KMP into their open-source library SQLDelight[49], which they also use in *Cash*

Cash App adopted KMP incrementally for shared logic.

[46] https://flutter.dev/showcase/sncf-connect
[47] https://flutter.dev/showcase/bmw
[48] https://kotlinlang.org/lp/multiplatform/case-studies/cash-app
[49] https://github.com/sqldelight/sqldelight

App. This step allowed the team to evaluate KMP in a controlled environment. Following its success, KMP was incrementally introduced to other shared application parts. By adopting KMP, the team maintained the native UI while sharing complex business logic and data models across platforms.

KMP improved team alignment and reduced development overhead.

This migration has brought the iOS and Android teams closer together. Developers retained their preferred native toolchains and minimized disruption to existing workflows but also benefited from the unified codebase reducing duplication. The team was able to improve consistency across platforms and reduce development overhead.

McDonald's adopted KMP incrementally for isolated components.

Another notable example of Kotlin Multiplatform adoption is McDonald's, which began using KMP in 2023 to address code redundancy in their Global Mobile App[50]. Rather than overhauling the entire application, McDonald's adopted KMP incrementally, focusing on isolated parts such as networking and data storage. This selective integration allowed the team to share functionality without compromising the responsiveness and native feel of the app. Leveraging in-house KMP libraries improved development speed in managing requirements like diverse markets.

[50] https://medium.com/mcdonalds-technical-blog/mobile-multiplatform-development-at-mcdonalds-3b72c8d44ebc

Chapter 3

Planning FLApp's Cross-Platform Transition

3.1 Overview of Features and Structure

Future Lab Aachen App (FLApp) is a mobile application that provides an interactive experience, guiding users to Aachen's scientific and historical landmarks. The app uses GPS to guide users to explore significant locations.

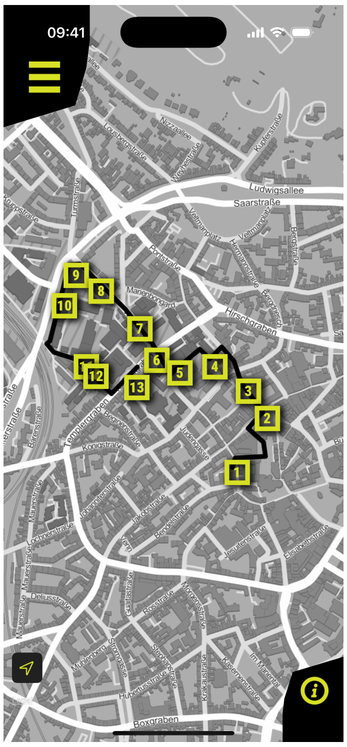
FLApp is a mobile guide for scientific topics in Aachen.

FLApp is build with a drill-down navigation pattern (Figure 3.2). The main screen consists of a map displaying all stations (POIs) throughout Aachen alongside a sidebar menu that lists these stations. Users can interact with the map or the list to select a station of interest, which takes them to a detailed overview page.

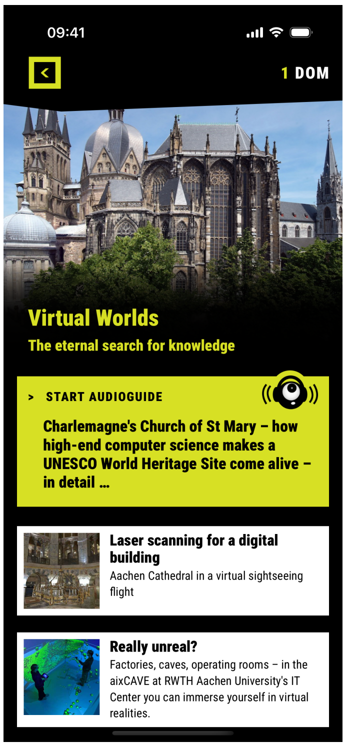
FLApp uses a drill-down navigation pattern.

Each POI's overview page (Figure 3.1b) provides rich multimedia content to enhance the user experience. An audio guide explains the significance of the site. Additional quotes, articles (Figure 3.1c), and visual elements such as images or videos complement this. Furthermore, AR features bring the content into reality, allowing users to inter-

Each POI has multimedia content (images, videos, audio & augmented reality)



(a) The user is guided along a route of all stations using GPS capabilities.


09:41

1 DOM

Virtual Worlds

The eternal search for knowledge

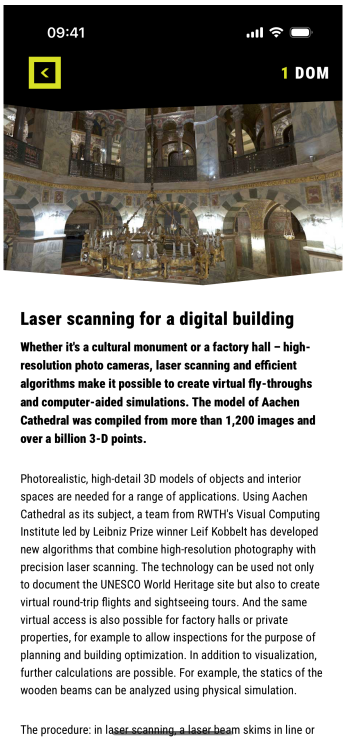
> START AUDIOGUIDE

Charlemagne's Church of St Mary – how high-end computer science makes a UNESCO World Heritage Site come alive – in detail ...

Laser scanning for a digital building
Aachen Cathedral in a virtual sightseeing flight

Really unreal?
Factories, caves, operating rooms – in the aixCAVE at RWTH Aachen University's IT Center you can immerse yourself in virtual realities.

(b) Each POI has an audio guide, article teasers, and sometimes AR content.


09:41

1 DOM

Laser scanning for a digital building

Whether it's a cultural monument or a factory hall – high-resolution photo cameras, laser scanning and efficient algorithms make it possible to create virtual fly-throughs and computer-aided simulations. The model of Aachen Cathedral was compiled from more than 1,200 images and over a billion 3-D points.

Photorealistic, high-detail 3D models of objects and interior spaces are needed for a range of applications. Using Aachen Cathedral as its subject, a team from RWTH's Visual Computing Institute led by Leibniz Prize winner Leif Kobbelt has developed new algorithms that combine high-resolution photography with precision laser scanning. The technology can be used not only to document the UNESCO World Heritage site but also to create virtual round-trip flights and sightseeing tours. And the same virtual access is also possible for factory halls or private properties, for example to allow inspections for the purpose of planning and building optimization. In addition to visualization, further calculations are possible. For example, the statics of the wooden beams can be analyzed using physical simulation.

The procedure: in laser scanning, a laser beam skims in line or

(c) Articles provide in-depth information about a topic using images and videos.

Figure 3.1: The three most important screens of the application are the map, the overview page of a point of interest, and the article screen.

act with 3D models or virtual representations of historical and scientific concepts.

FLApp used Objective-C with UIKit for iOS and Kotlin with Android View.

FLApp was initially developed and released in 2017. The iOS and Android apps were implemented as native applications using the iOS SDK (UIKit and Objective-C) and Android SDK (View System and Kotlin). Both implementations use advanced platform features like Core Location, ARKit (iOS), and AltBeacon[1] or SceneForm[2].

1 https://github.com/AltBeacon/android-beacon-library
2 https://github.com/google-ar/sceneform-android-sdk

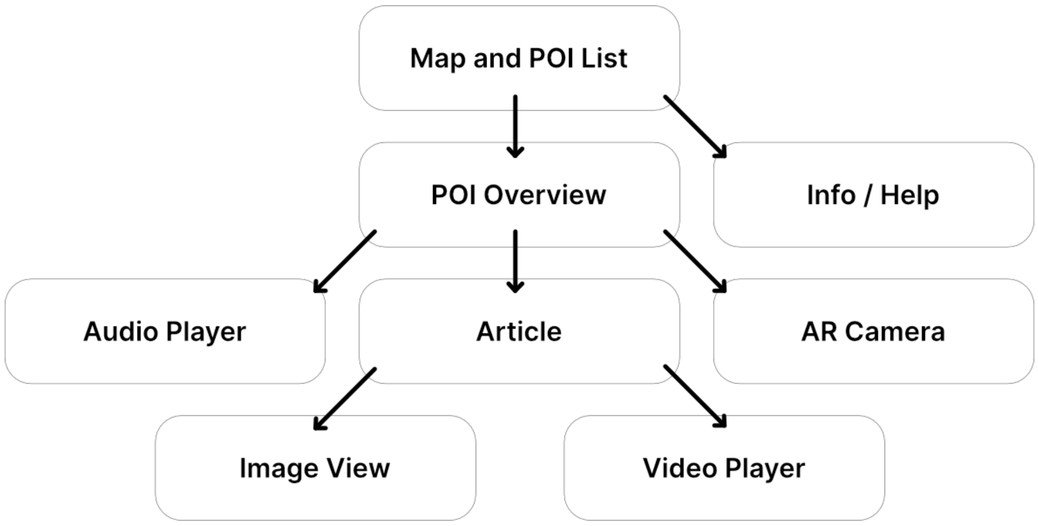


Figure 3.2: FLApp uses a drill-down navigation pattern to display information about POIs and related topics. A couple of utility screens are used to provide additional information.

3.2 Motivation for the Cross-Platform Transition

FLApp has accumulated significant technical debt. Over time, the app's technology, architecture, and data model have become increasingly difficult to maintain. In particular, the iOS implementation has remained mostly unchanged since 2017, contributing to inconsistencies and maintenance challenges. To address these issues, we have decided to technically overhaul the app using cross-platform technology, aiming to improve maintainability and eliminate inconsistencies.

FLApp built up technical debt that we want to resolve.

3.2.1 Problems

Our initial code review and past experiences with content updates and bug fixes revealed a key issue: the iOS and Android apps exhibit inconsistent behavior, even with identical inputs. This fact complicates testing and introduces unpredictable results.

Problem 1: Maintenance of two codebases

Problem 2: Tight coupling, lack of uniform architecture, and poor separation of concern

The current architecture of the apps further compounds these challenges. The iOS implementation, based on a tightly coupled Model-View-Controller (MVC) pattern, directly loads data within view controllers, preventing data model flexibility. In contrast, the Android app employs the MVVM pattern but relies on a complex parser class throughout. This parser violates the single responsibility principle and introduces significant maintainability challenges. Both codebases suffer from tight coupling and a lack of clear separation of concerns.

Problem 3: Onboarding new developers

Having two native implementations presents significant challenges for new developers joining the project. The outdated Objective-C codebase in the iOS app and inconsistent implementations across platforms create a steep learning curve. New developers must work on two very distinct architectures and require adjustments for each platform.

3.2.2 High-Level Goals

High-level goals help us select the CPT.

To guide the selection of a cross-platform technology (CPT) for FLApp, we established the following high-level goals derived from the previously identified problems. These goals served as critical criteria during the technology evaluation process.

Goal 1: No regression regarding features and user experience.

Our primary objective is to preserve all existing features and the user experience without regression. The new technology must support critical capabilities such as Augmented Reality (AR) and tracking iBeacons at the stations.

Goal 2: Remove duplication to improve maintainability.

We aim to eliminate code duplication between platforms wherever feasible. By that, we want to ensure a single source of truth and consistent behavior across platforms.

Goal 3: Introduce a uniform app architecture.

We want to establish a unified app architecture across both platforms to reduce cognitive load for developers and enhance maintainability. A consistent architecture will streamline development and ensure coherence in the codebase, leading to a more efficient development process.

3.3 Cross-Platform Framework Selection

One of the key tasks before the actual migration of FLApp to a cross-platform technology is the choice of a suitable development framework.

We have to choose a suitable CPF for the migration.

In section 2.1 and section 2.2, we already introduced four cross-platform options from different categories and looked at similar migrations. This prior work is our basis for understanding the potential advantages and challenges of adopting a cross-platform framework.

We base our decision on previous research.

However, the related work reflects individual experiences that may not directly apply to our specific application. We take valuable insights from these studies, but they sometimes align differently with our project's requirements and context, making a detailed analysis for FLApp necessary.

We need a specific analysis despite our related work.

3.3.1 Decision Frameworks

We researched important criteria to consider when selecting a suitable CPF for FLApp. While we also had several important criteria in mind before our research, this approach ensured we respected all the significant aspects during our selection process.

We research important aspects to select the CPF.

Rieger and Majchrzak [2019] proposed a Decision Framework (DF) tailored for evaluating cross-platform frameworks on a per-project basis. This framework employs 33 criteria, organized into four key perspectives: infrastructure, development, app, and user (Figure A.1). Each framework is scored on a scale from 0 (entirely dysfunctional) to 5 (fully functional) across these categories. Additionally, the DF supports company- or project-specific configurations through customizable weight profiles, making it adaptable to diverse project requirements. An evaluation study demonstrated its applicability by assessing five cross-platform frameworks, offering a structured approach for selecting the most suitable option for specific projects.

Rieger proposed a Decision Framework with 33 criteria.

Lachgar and Khachouch introduced other Decision Frameworks.

Further research on Decision Frameworks includes the work of Lachgar et al. [2022], who introduced a more complex DF based on multi-criteria decision-making (MCDM) methods, and Khachouch et al. [2020], who presented nine guiding questions for selecting a general technology approach (e.g., Native, Web, Hybrid, Cross-platform, Cloud-based, or Merged) rather than focusing on specific frameworks.

Rieger highlights challenges in evaluating frameworks due to scientific research.

However, Rieger and Majchrzak [2019] also highlight challenges in evaluating frameworks using literature and scientific studies: Comparisons often involve a limited subset of frameworks. Furthermore, diverse research objectives can lead to inconsistent or even contradictory findings. This inconsistency is evident even in seemingly objective, quantitative performance measures, complicating the evaluation process and demanding caution in interpreting such results.

Non-standardized CPF research prevents reliable decision frameworks.

This lack of consistent and up-to-date scientific research makes developing and maintaining objective and reliable decision frameworks difficult. The use of various studies with differing methodologies further compounds the problem.

Rieger's evaluation is outdated and excludes important CPF candidates.

In our context, we cannot use the assigned scores of Rieger and Majchrzak [2019]. Although they scored React Native and PWAs, their evaluation reflects the state of the technology in 2019, making some criteria potentially inaccurate for today's state. Their work also does not include Flutter and Kotlin Multiplatform because these frameworks were unpopular and unstable in 2019.

We use the DF's criteria for our selection process.

Using the DF in the intended way would require us to assign new scores based on a systematic literature review. However, this was not the focus of our work, so we decided to use the Decision Framework solely to identify the key criteria relevant to our decision regarding FLApp rather than conducting a new detailed general scoring.

3.3.2 Criteria-Based Comparison in the Context of FLApp

Besides our high-level goals, we have extracted aspects relevant to FLApp from the decision framework Rieger and Majchrzak [2019] and selected them for our selection process. Table 3.3 summarizes the key criteria we extracted from their work relevant to FLApp's CPF choice.

We established goals and criteria for selecting a cross-platform framework.

Perspective	Criteria
App perspective	Hardware Access (A1) Platform Functionality (A2) Input Heterogeneity (A4) Application Lifecycle (A6) Robustness (A9)
Infrastructure perspective	Target Platforms (I2) Distribution Channels (I4) Internationalisation (I6) Long-term Feasibility (I7)
Development perspective	Developer Environment (D1) Preparation Time (D2) UI Design (D5) Testing (D6) Maintainability (D9) Extensibility (D10) Custom Code Integration (D11)
User perspective	Look and Feel (U1) Performance (U2)

Table 3.3: We extracted all criteria from [Rieger and Majchrzak, 2019] relevant to FLApp's framework decision.

App Perspective

In our decision-making process, we prioritized several key criteria. Hardware Access (A1) and Platform Functionality (A2) play significant roles for FLApp, especially considering our need for ongoing support for Bluetooth Beacons

Hardware access and technical functionalities like AR and Bluetooth were prioritized.

and Augmented Reality capabilities, which are integral to the application's functionality.

Background beacon ranging and offline support were identified as critical requirements.

Application Lifecycle (A6) was important as we needed to enable background ranging for beacons, ensuring smooth operation even when the app is not in the foreground. Furthermore, we need Robustness (A9), particularly for offline support and handling situations like missing device permissions, such as camera access.

We focused on meeting technical requirements for specific features.

As the first step in our decision-making process, we wanted to exclude technologies that could not meet our technical requirements for the current features (Goal 1, A1, A2, and A9).

PWAs lack Bluetooth and AR capabilities critical to project goals.

After initial research, we quickly ruled out the use of Progressive Web Apps. PWAs cannot support Bluetooth beacons with the same quality as our current implementation (A1). Web Bluetooth Scanning required for using beacons as geomarkers is still a draft proposal[3]. For FLApp, we would also need background-ranging support, allowing the user to receive push notifications about the near station even when the app is not actively running (A2). Furthermore, supporting augmented reality (AR) would likely have been problematic. While the relatively new WebXR Device API[4] exists, it is still in the experimental stage and is not consistently available across devices (A2, A4).

Narrowed choices to React Native, Flutter, and KMP.

Maintaining these functionalities is a core objective of our cross-platform transition, so we were compelled to remove Progressive Web Applications as a viable candidate. This left us with three remaining candidates: React Native, Flutter, and Kotlin Multiplatform.

[3] https://webbluetoothcg.github.io/web-bluetooth/scanning
[4] https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API

Infrastructure Perspective

Aspects like Target Platforms (I2), Distribution Channels (I4), and Internationalisation (I6) are well-defined for FLApp. We needed a CPF that supports iOS and Android. The app would still be distributed via the App Store and Google Play, and we needed to support localization in English and German. As all our remaining CPF candidates support these criteria, we consider them irrelevant to our decision.

All candidates have support for our required platforms, distribution, and localization.

Long-term feasibility (I7) is a critical factor guiding our decision. We seek a framework that remains compatible with new operating system features, receives active maintenance, and ensures timely resolution of bugs and issues. We assessed this criterion by examining the framework's maturity, stability, and activity, as recommended by Rieger and Majchrzak [2019].

Our framework should be mature and supported for a long time.

React Native, Flutter, and KMP are all stable release frameworks that have been actively developed for several years. The previously discussed case studies demonstrate that all three frameworks have proven themselves in production environments, even for large-scale enterprises.

React Native, Flutter, and KMP are stable frameworks.

Kotlin Multiplatform reduces technological lock-in by leveraging Kotlin, Android's recommended programming language[5], and compiling to native LLVM code. This design allows KMP modules to be more easily replaced with native code in end-of-life (EOL) scenarios, offering a flexibility advantage over React Native or Flutter.

KMP offers reduced technological lock-in.

This also applies to Compose Multiplatform, built on Android's declarative UI framework, Jetpack Compose. We believe migrating from Compose Multiplatform back to Jetpack Compose would require relatively little effort for Android. The fact that transitioning from Jetpack Compose

Compose Multiplatform provides an easier migration path for Android.

5 https://developer.android.com/kotlin/first

to Compose Multiplatform[6] can be achieved in just a few steps supports this[7].

Development Perspective

Development tooling is a critical factor.

Several aspects are important when considering the development perspective. The Developer Environment (D1) and tooling are critical for effective implementation and usability.

Flutter and React Native integrate well with popular IDEs, and KMP mainly uses Android Studio.

As described earlier, Flutter and React Native provide plugins for integration into popular IDEs such as VS Code, IntelliJ, and Android Studio. They also offer developer tools for advanced debugging needs. Kotlin Multiplatform (KMP) development is currently primarily done in Android Studio. For all three frameworks, implementing native interoperability typically requires working within platform-specific IDEs, such as Xcode for iOS and Android Studio for Android.

JetBrains' expertise may give KMP an edge in the future.

Kotlin Multiplatform has strong potential to lead in IDE support, largely thanks to JetBrains' significant influence and expertise. As the creators of IntelliJ and other commercial IDEs, JetBrains develops the foundation for Android Studio and plays a central role in Kotlin's ecosystem.

JetBrains plans KMP-specific IDE.

Although tools like the KMP Plugin for Android Studio are already available, JetBrains has announced plans to introduce a KMP-specific IDE[8] with integrated language support for Kotlin and Swift. This new IDE may require a paid license for commercial use. However, we are confident that the new IDE will not be required and that the combination of Android Studio and Xcode will continue to be sufficient.

[6] https://touchlab.co/compose-multiplatform-transition-guide

[7] Unfortunately, we were unable to test the transition from Jetpack Compose to Compose Multiplatform in FLApp, as the existing Android implementation still uses the traditional View System.

[8] https://blog.jetbrains.com/kotlin/2024/10/kotlin-multiplatform-development-roadmap-for-2025/#tooling

JetBrains wields considerable leverage through its involvement in the Kotlin Foundation, where it collaborates with Google in key positions on the Board[9] and in the language design process. This dual role – shaping both the language and its tooling – allows JetBrains to create a highly integrated and coherent ecosystem. For developers, this "ownership of the stack" may translate into significant advantages in terms of integration and usability. However, this can also cause problems regarding dependence on a single provider.

JetBrains leverages a strong position in Kotlin's ecosystem.

We concluded that the Development Environment (D1) was sufficient for all three candidates, with slight advantages for KMP. JetBrains' natural involvement with IDEs will allow them to improve the ecosystem further in the future.

Developer environment is sufficient for all candidates with slight advantages for KMP.

Preparation Time (D2) was also a more significant concern. This includes factors like entry barriers, required language skills, setup on the local machine, build tools, documentation, and literature. The selected framework should be easily adaptable for developers since also computer science students at our chair will continue to maintain this project.

Preparation time is crucial for a project like FLApp.

React Native primarily uses the widely adopted JavaScript language, similar to the React web framework. This familiarity allows experienced React developers to get started with React Native quickly. While JavaScript is not type-safe, developers can use TypeScript as an alternative. Kotlin is also already widely used, particularly for Android development, and shares similarities with other object-oriented programming languages like Java and Swift. On the other hand, Dart is a relatively lesser-known language primarily used for Flutter development. As a result, most developers are unlikely to have prior experience with it (Dart is position 33, Kotlin at 23, and JS at 6 in the TIOBE Index in 2024[10]).

JavaScript and Kotlin are popular, but Dart is only used for Flutter development.

9 https://kotlinfoundation.org/structure
10 https://www.tiobe.com/tiobe-index/

Doctor commands check the correct setup of the needed tools.

All candidates provide so-called doctor commands checking the local setup of all required technologies and providing help for the most common errors (flutter doctor[11], react-native doctor[12], and kdoctor[13]). They also offer configurable scaffoldings that can be used to set up new projects.

Deploying to an iOS Simulator or a real device requires macOS.

Generally, deploying and testing the iOS app will always require using macOS. However, a few cloud-based services offer running the apps for testing purposes.

KMP primarily uses Gradle; React Native uses NPM, and Flutter has a CLI.

Kotlin Multiplatform currently relies primarily on Gradle as its build tool. While Gradle may present a learning curve and potential drawback for non-Android developers, the configuration process is expected to become more straightforward with the introduction of Amper[14], a Gradle-compatible build tool currently in beta. React Native uses NPM as its package manager and build tool. Flutter uses its package manager *pub*, integrated with Dart and custom CLI tools for building, testing, and shipping the app.

There is limited research comparing learning experiences, but all seem adaptable in a reasonable timeframe.

We did not find up-to-date scientific research comparing the three frameworks regarding learning experience. However, Zhang [2024] found that they could teach Flutter to computer science students through six-week online courses with self-directed learning to a level where they can start implementing apps and features on their own. We did not find similar work for the others, but we think students and experienced developers can learn basic React Native and KMP in a similar timeframe.

Native integration requires more profound platform knowledge.

For more advanced integrations, developers will always need more in-depth or broad knowledge about the development of the target platform, as bridging into the native SDK may be needed there. This may also be a problem for KMP when not using Compose Multiplatform but

[11] https://www.dhiwise.com/post/flutter-doctor-command-a-vital-tool-for-developers
[12] https://reactnative.dev/blog/2019/11/18/react-native-doctor
[13] https://github.com/Kotlin/kdoctor
[14] https://blog.jetbrains.com/amper/2024/11/amper-update-november-2024

natively implementing the UI, as this requires knowledge about SwiftUI/UIKit and Jetpack Compose/View System.

Regarding the UI implementation methods (D5), all frameworks support defining UI declaratively and follow a state-driven approach. Nevertheless, there are generally two different approaches among the frameworks: React Native and Flutter are designed to use a shared frontend, which makes mixing and matching cross-platform interfaces with native components more complex (see D10 and D11). Still, building a UI with React Native is easy for developers already familiar with React, as it uses almost the same syntax.

All frameworks have options to manage UI with state declaratively.

KMP takes a different approach by making the adoption of the interface library Compose Multiplatform (CMP) entirely optional and explicitly leaving developers the choice whether they want to use a shared interface technology[15]. Another advantage of CMP is that the Android implementation is based on the Android Jetpack Compose UI library. While this does not apply to FLApp, it makes adoption for existing native applications using Jetpack Compose much easier. On iOS, CMP can be gradually adopted by displaying a Composable (Composes' name for a view) inside of a `UIViewController`.

KMP makes adoption of Compose Multiplatform optional.

All three frameworks provide capabilities for Testing (D6). Flutter supports unit tests, widget tests, and integration tests[16]. React Native supports unit tests, integration tests, component tests, and end-to-end tests. KMP uses Kotlin Test[17] frameworks for unit testing. When using native UI, native UI tests (iOS) and Instrumentation tests (Android) are recommended. Compose Multiplatform UI Testing[18] is currently in experimental state. While the frameworks use different terms for specific testing approaches, they all fulfill our testing needs (mostly unit tests).

All frameworks support various testing methods.

[15] KMPs value proposition was initially focused on sharing business layer code. CMP went Alpha in 2021 after the first versions of KMP were already released.
[16] https://docs.flutter.dev/testing/overview
[17] https://kotlinlang.org/api/core/kotlin-test
[18] https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-test.html

Maintainability is crucial for long-term success.

The long-term success of our CPF transition depends on its Maintainability (D9), which is closely related to Long-term Feasibility (I7). All three frameworks would offer better maintainability after adoption than the current implementations, which is the main reason for our transition.

Extensibility and custom code integration are key for our application.

Extensibility (D10) and the ability to integrate custom code (D11) are critical for implementing features not directly supported by cross-platform frameworks (CPFs), significantly when leveraging third-party libraries or accessing platform-specific functionality.

Flutter uses Platform Channels for native integration with some limitations.

Flutter provides native integration via Platform Channels[19], enabling communication between Dart code and platform-specific APIs. This approach supports access to native functionality but comes with some limitations. For example, Flutter's documentation highlights that Platform Views on Android offer two implementations, each with trade-offs in terms of performance and visual fidelity[20]. Additionally, Flutter's reliance on its custom rendering engine can make integrating native UI components more complex.

React Native uses Native Modules for integration, affecting performance in some cases.

React Native uses Native Modules[21] to connect JavaScript with platform-specific code, offering flexibility for integrating native functionality and third-party libraries. Its architecture supports communication with native systems. However, performance may be affected in cases where frequent communication between the JavaScript layer and native components is required [Oliveira et al., 2023].

KMP is designed for native interoperability and incremental adoption.

KMP is explicitly designed for incremental adoption and native interoperability, making it especially suited for integrating into existing native apps. Its ability to work directly with platform-specific code allows developers to retain or implement native functionality without significant overhead. This is especially good for incremental migration approaches or advanced native integration. We think that KMP makes mixing shared and native code much eas-

[19] https://docs.flutter.dev/platform-integration/platform-channels
[20] https://docs.flutter.dev/platform-integration/android/platform-views
[21] https://reactnative.dev/docs/turbo-native-modules-introduction

ier due to the design philosophy of expected and actual implementations being similar the interface/implementation paradigm in object-oriented languages.

User Perspective

Lastly, the user perspective is equally important as the development perspective. The application's Look and Feel (U1) and Performance (U2) influence its overall success and user satisfaction.

User perspective is equally important.

We discussed parts of Look and Feel (U1) already during UI Design (D5) and are confident all (UI) frameworks could be used to implement our current required user interface.

All UI frameworks can implement the necessary interface.

Perceived performance (U2) is also critical from the user's perspective. Oliveira et al. [2023] built multiple benchmarks in Flutter, React Native, Ionic, and as a native Android app and assessed their performance. Their results show that each framework introduced an overhead, but Flutter was the most performant for CPU-intensive tasks due to cross-compilation while using more memory. However, other studies showed that such overhead is not always perceivable by the user, indicating little to no degradation of user experience [Angulo and Ferre, 2014; Axelsson and Carlström, 2016].

There is a slight performance overhead for React Native and Flutter.

Skantz [2023] evaluated the performance of KMP on iOS writing benchmarks for networking (HTTP and JSON parsing), database (SQLite), and parts of the Computer Language Benchmarks Game (CLBG). They found that using Kotlin was sometimes even faster than the native implementation, with the drawback of higher memory and CPU usage. Additionally, their results show correlations between the garbage collection cycles of KMP and profiling patterns of memory consumption and CPU usage. Overall, their research showed almost no performance degradation on iOS and argued that KMP uses Kotlin like native Android development, so no differences should be noticeable.

Skantz's study evaluated KMP's performance on iOS and found it comparable to native.

All candidates can provide the necessary performance.

Performance-wise (U2), we think all candidates will be able to fulfill FLApp's needs. Despite the more resource-intensive maps and especially AR capabilities, the app only presents information like text, images, videos, and audio. For AR, we could always incorporate native technology – even with React Native and Flutter.

3.3.3 The Final Decision

All three frameworks could implement FLApp from scratch.

Overall, it would likely be possible to implement FLApp from scratch using any of the three technologies. Nevertheless, both Flutter and React Native may require either external libraries or interoperability with native code to support features like augmented reality and beacon capabilities.

Incremental migration is more suitable than a reimplementation.

However, we concluded that an incremental migration might be a better strategy as we realized FLApp's problems mainly relate to the data and business layer rather than the user interface. As stated by Deka et al. [2017], translating a design into code can be lengthy. So we concluded an entire reimplementation would be a big undertaking.

Reimplementation risks more (unnoticed) regressions.

Likely, a reimplementation would also bring some regressions – even if just some UI details were not implemented like before. This seemed unacceptable because other stakeholders were also involved with the design and in the general project[22].

Case studies highlight challenges in complex incremental migrations.

While the case studies in subsection 2.2.2 revealed both the advantages and disadvantages of incremental migrations, their challenges mainly occurred in large, complex applications. These projects had hundreds of screens, unified design systems, large development teams, and ongoing feature development.

Advantages of incremental migration outweigh the problems for FLApp.

Since our app differs from these scenarios, we concluded that the advantages of an incremental, step-by-step migration outweigh the problems presented in the case studies. This approach could allow us to ensure the app remains functional throughout the migration process. It is

[22] https://futurelab-aachen.de/en

also much more efficient in terms of time and effort to not start from scratch.

KMP best supports incremental migration.

The only framework among our candidates that effectively supports this use case is Kotlin Multiplatform. Choosing KMP would allow us to save significant effort and avoid potential challenges when implementing the user interface. Additionally, it would provide the flexibility to integrate new screens using Compose Multiplatform in the future or gradually replace existing ones.

Following this, we developed a migration plan and implemented a proof-of-concept feature to validate our approach.

Chapter 4

Migrating FLApp to Kotlin Multiplatform

Based on our decision to migrate incrementally, we set the primary migration objective to maintain fully functional implementations at all times, preserve all existing features, and prevent any regressions. Due to the absence of Unit and UI tests, we relied on manual testing to verify the progress. During the migration, we then added some Unit tests for new implementations.

Our primary objective during migration was to keep the apps shippable.

The following sections detail the migration of FLApp to KMP, covering our process, reasoning, and experiences. We highlight the critical decisions and workflows, especially those specific to KMP. Due to the scope of this migration, we cannot discuss every nuance of the transition.

We cannot explain every detail of the migration and code.

We took advice and inspiration from some case studies like the native-to-KMP migration at Cash App[1], the experience at Worldline[2], and the introduction of KMP at 9GAG[3].

We took some inspiration from KMP case studies.

We designed a multi-step migration strategy to address FLApp's unique requirements and implementation: To migrate FLApp effectively, we designed a multi-step strategy

We planned a multi-step migration strategy.

[1] https://kotlinlang.org/lp/multiplatform/case-studies/cash-app
[2] https://blog.worldline.tech/2022/01/26/kotlin_multiplatform.html
[3] https://raymondctc.medium.com/adopting-kotlin-multiplatform-mobile-kmm-on-9gag-app-dfe526d9ce04

that addressed its unique requirements. First, we consolidated the iOS and Android implementations into a single Kotlin Multiplatform (KMP) project. This also involved migrating the iOS codebase from Objective-C to Swift. While KMP would later replace some of the Swift code, our initial goal was to run both iOS and Android in Swift and Kotlin, making it easier to abstract the shared logic. Next, we planned to implement GPX parsing as a proof of concept using KMP. This would then be followed by moving all "non-Android" code from the Android module into a shared module. Finally, the strategy included redesigning the data model and app architecture to align with a new data format and eliminate duplicated code, inconsistencies, and tight coupling.

4.1 Setup of KMP

We manually set up KMP for our existing implementations.

When starting a new KMP project, developers can use the Kotlin Multiplatform Wizard[4] or use the KMP plugin[5] for Android Studio. For FLApp, we already have Android and iOS implementations, so we created a project with the KMP Wizard as a template for manually setting up KMP in our existing project.

4.1.1 Repository Setup

KMP works with different repository setups.

KMP works in a wide variety of repository setups. The default KMP template sets up the code for iOS, Android, and possibly other platforms in one (mono) repository. However, the Android+Shared and iOS code can also be located in different repositories or distributed even further across different repositories (distributed KMP libraries).

[4] https://kmp.jetbrains.com
[5] https://plugins.jetbrains.com/plugin/14936-kotlin-multiplatform

We chose a monorepo setup because FLApp is relatively simple in features and complexity. A monorepository removes management overhead and improves developer experience. According to Jaspan et al. [2018], using a monorepo setup also facilitates creating unified APIs and encourages better software architecture, which is precisely our goal. Moreover, it simplifies managing build tools like dependency versions.

A monorepo setup reduces complexity and streamlines the build process for FLApp.

4.1.2 Module and Folder Setup

The default scaffolding uses a *single shared module* approach, where a single KMP module named `shared` is utilized to share code between iOS and Android. However, it is also possible to create individual KMP modules to separate code for specific features or domains[6].

KMP allows different module setups.

A single shared module is a great starting point because it reduces cognitive load and simplifies starting with KMP. However, according to KMP's documentation, compile time may increase as the code in the module grows.

Single shared module reduces cognitive load.

Utilizing multiple modules enhances modularity and scalability while promoting code encapsulation. This approach encourages developers to consider their code structure carefully and facilitates better visibility control. However, it can also complicate dependency management and Gradle setup.

Using multiple modules may improve encapsulation but complicate setup.

[6] https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-project-configuration.html#module-configurations

```
FLApp
├── androidApp
│   ├── build.gradle.kts
│   └── src
├── iosApp
│   └── ...
└── shared
    ├── build.gradle.kts
    └── src
        ├── androidMain
        ├── commonMain
        ├── commonTest
        └── iosMain
```

Figure 4.1: Excerpt of the Directory Structure.

Excursus: *Umbrella Framework*

UMBRELLA FRAMEWORK:
JetBrains recommends using an umbrella module to bundle multiple shared modules for integration into iOS. The Kotlin/Native compiler cannot de-duplicate bundled dependencies across multiple exported Apple Frameworks.
So, not using an umbrella module would lead to a bigger app size. Furthermore, it improves the resulting artifact and eliminates inconsistencies between dependency versions.

We moved the iOS implementation into the Android repository.

To streamline the onboarding process for new developers and adhere to standard monorepo conventions, we decided to integrate the iOS implementation into the existing Android repository. Although it is possible to configure the folder structure manually, we followed the default directory structure (Figure 4.1) to maintain consistency and simplicity across the project.

4.1.3 Build Tools: Gradle

KMP utilizes Gradle modules on Android and compiles frameworks for iOS.

KMP uses Gradle as the primary build tool and integrates right into existing Android projects because it also uses Gradle. The iOS toolchain does not share the same build tools and language, so KMP uses the Kotlin/Native compiler to create Apple frameworks. A custom Xcode build step is being used to integrate the Apple framework. This step calls Gradle to generate a binary of the framework that will be linked to the project. The other option is using a package manager like CocoaPods or Swift Package Manager, with the framework artifact hosted on a remote registry. However, this requires the framework to be compiled separately and uploaded to the registry.

KOTLIN/NATIVE COMPILER:
The Kotlin/Native compiler is based on LLVM and enables the creation of native, self-contained binaries that run without an additional runtime or virtual machine[7].

Excursus:
Kotlin/Native compiler

For that reason, both Xcode (Xcode Command Line Tools) and Android Studio (Gradle) need to be installed on the developer's machine to develop KMP projects.

Xcode and Android Studio are required for building KMP apps.

JetBrains is actively working to create a more accessible build tool for KMP and Kotlin development called Amper[8]. Amper will simplify setting up and maintaining the build tools of a KMP app. It is designed to be used on top of Gradle or as a standalone tool. With that design, setting up a new module using Amper in an existing Gradle project should be easy. However, we decided against using Amper for FLApp as it was only in the beta phase.

JetBrains develops Amper build tool to simplify and replace Gradle in KMP.

Before starting the KMP migration, we migrated the Gradle build files using Gradle Version Catalogues. These are recommended in KMP to ensure that each module uses the same dependency version.

We switch to use Gradle Version Catalogs.

We then started the setup of the KMP module: When embedding the new `shared` module into the existing Gradle setup, we had to add it into the `settings.gradle.kts` to register the module in the root Gradle project (Listing B.2).

We added a new shared module to our Gradle project.

In the Gradle file of the `shared` module, it is necessary to apply the Kotlin Multiplatform Gradle plugin.

The Kotlin Multiplatform Gradle plugin is applied to the shared module.

[8] https://github.com/JetBrains/amper

```
kotlin {
    iosArm64() // Targeting 64-bit iOS
    iosX64() // Simulator on Intel macs
    iosSimulatorArm64() // Simulator on Apple Silicon macs
    androidTarget() // Android target
    // also available: JVM, JS, and Wasm
}
```

Figure 4.2: This configuration code is used in `shared/build.gradle.kts` to specify the multiplatform targets of the Gradle module `shared`.

This plugin automatically adds necessary dependencies to, e.g., the platform-specific (iOS and Android) parts of `kotlinx.coroutines` needed for asynchronous code.

We specify target platforms in the module-level build script.

In the module-level build script `build.gradle.kts`, we specify the target platforms for which our shared module should compile. We can specify Android, iOS, and JVM platforms and edit their settings (Figure 4.2).

KMP modules use source sets to organize source files with their targets and dependencies.

We need to define source sets for each KMP module. Source sets [9] are collections of source files, each with its own set of targets, dependencies, and compiler options. Each source set has its folder and can be configured further using the `sourceSets` Gradle attribute, allowing specification of the source set specific dependencies. Also, source sets for testing can be configured, e.g., `commonTest` contains tests for the `commonMain` source set (Listing B.3).

The compiler collects code hierarchically from source sets for target platforms.

When compiling for a target platform, the code across all involved source sets is collected hierarchically (see Figure 4.3). For instance, an iOS build targeting an actual device would include code from the `commonMain`, `appleMain`, and `iosMain` source sets (Figure 4.3). This hierarchical approach allows for fine-grained customization between targets, enabling the definition of specific compiler options and especially dependencies for each source set.

[9] https://kotlinlang.org/docs/multiplatform-discover-project.html#source-sets

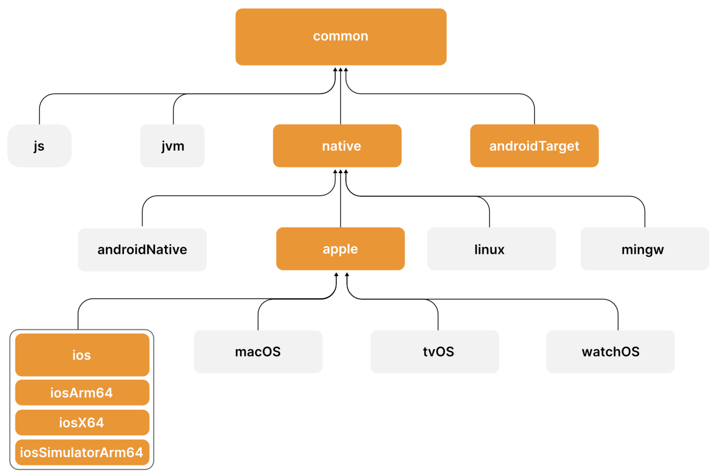


Figure 4.3: This hierarchical tree is used for building a specific KMP module. If the `iosArm64` is requested to be built, all sources from `iosArm64`, `ios`, `apple`, `native`, and the `common` source set are used to compile the module. This graphics is adopted from the KMP documentation.

For FLApp, we use the default Android and iOS targets (`iosArm64`, `iosX64` and `iosSimulatorArm64`) to allow compilation for all possible targets (Simulators and real devices).

The compiler collects code hierarchically from source sets for target platforms

4.1.4 Build Tools: iOS Integration

Now that we have specified our compilation targets and can use common code from our `shared` module on Android, we still need to include the framework in our Xcode project on iOS.

We need to include the shared framework in the iOS app.

First, we need to set up our iOS target in Xcode to compile our shared framework each time we build the iOS project. Therefore, we add a *Run Script Phase* (Listing 4.1), which runs a Gradle task and creates a `.framework` in the output directory at `shared/build/xcode-frameworks`.

We use a Run Script Phase to compile the shared framework each build.

```
cd "$SRCROOT/.."
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```

Listing 4.1: Run Script for the iOS Target

Framework Search Paths are configured to locate the Shared framework.

Lastly, we need to set up the *Framework Search Paths* in FLApp's build settings to instruct Xcode where to search for our `Shared` framework. It needs to be set to `$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)` to resolve to the correct framework. Then, in every Swift source file, we can import our `Shared` framework and use the bridged classes.

4.2 Migration From Objective-C to Swift

We migrated the iOS app to Swift to improve Kotlin interoperability and developer experience.

As part of the migration plan, we then migrated the iOS app from Objective-C to Swift, as Kotlin and Swift are more similar languages compared to Objective-C, providing a similar development experience [Schultes, 2021]. Furthermore, we want to ensure easier onboarding for new developers and switch to a more modern language.

A Bridging Header imports Objective-C into Swift during migration.

In order to incrementally convert Objective-C code to Swift, we introduced a Briding Header[10], which allows importing Objective-C code in Swift.

We migrated entry points first, then every class, and verified the success manually.

At first, we migrated the entry point `main.m` and `AppDelegate` to Swift and added all referenced Objective-C classes to our Bridging Header. We progressed the migration class by class and manually verified the migration's success. We avoided creating circular dependencies between Objective-C and Swift code, as the compiler cannot process imported source files in the bridging header using new Swift code. Sometimes, we had to reconnect the storyboard and xib user interface files with the code. Occasionally, we could also remove old code that was no longer used. However, we did not refactor and translate the code line by line during that step.

[10] https://developer.apple.com/documentation/swift/importing-objective-c-into-swift

4.3 Refactoring and Implementing Features With KMP

Once we had completed migrating the iOS implementation to Swift, we could start using KMP to integrate the first shared code. To get started, we decided to extract a small, not too complex part of the business logic and try out KMP for the first time.

4.3.1 Implementing a Shared GPX Parser

Parsing GPX served as a proof-of-concept for a KMP implementation.

We decided to use GPX parsing, which displays the tour route on the app's map. GPX is an XML-based standard that simplifies the exchange of GPS data (waypoints, routes, and tracks)[11]. In the pre-KMP version of FLApp, two separate implementations existed for iOS and Android. The Android version used `javax.xml` to iteratively extract all found track points (`trkpt`) and their positions as coordinates directly from the XML document. The iOS implementation used the `NSXMLParser` to read the GPX file into an object structure consisting of `GPX`, `GPXEntry`, `GPXLink`, `GPXTrack`, `GPXWaypoint` and later extracted the coordinates from the data structure.

The differences in implementation between the platforms can quickly lead to problems.

In both implementations, parsing the existing GPX route worked without any problems. However, this example illustrates how even simple features can be implemented quite differently across platforms. It is easy to see that this can quickly lead to bugs and inconsistencies. With additional requirements, such as displaying several routes or other points, two very different implementations have to be adapted.

We use a KMP library for parsing the underlying XML.

For our KMP implementation, we use the KMP library `XmlUtil`[12] for parsing the GPX. This library is based on the `kotlinx.serialization` package, similar to the `Codable` protocol in Swift. All entities and relationships between them are implemented as data classes (Kotlin equivalent

[11] https://www.topografix.com/gpx.asp
[12] https://github.com/pdvrieze/xmlutil

to Swift structs) according to the GPX specifications and are annotated with the respective XML element names via `@XmlSerialName`.

Additional unit tests ensure expected functionality.

In order to ensure functionality during development and in the future, we have written unit tests in the `commonTest` folder using the Test-Driven-Development (TDD) approach. We used the first-party `kotlin.test` framework to implement the unit tests.

We implemented a parser decoding a GPX String and returning a typed GPX entity.

The newly implemented `GPXParser` decodes a GPX String and returns a typed `GPX` entity, which can then be used at higher levels of abstraction. In our case, we first integrated this parser directly into the current map on iOS and Android to test its functionality, which worked without any additional adjustments.

Our implementation could also be used as an external library in other projects.

We deliberately implemented GPX parsing functionality beyond the immediate requirement of parsing track points. This approach allows our KMP-based GPX parser to be quickly packaged as an external library, making it reusable in other projects with more extensive or varied requirements.

The GPX library can be distributed for KMP or as an XCFramework.

To bundle this as a framework, the relevant source files could be extracted into a separate repository using the Multiplatform Library Template[13] and published to a registry like Maven Central. Additionally, with KMMBridge[14], a standalone XCFramework can be compiled, enabling seamless integration into native iOS projects.

4.3.2 Sharing Files and Resources

The `RouteService` abstracts away loading from disk, parsing the GPX file and returning coordinates.

During our initial test, we loaded the GPX Strings from the files in native code and extracted the coordinates required for the route separately from the returned GPX object. Now, we continued by implementing a `RouteService`, which is responsible for loading the route: The `RouteService` loads the GPX file from the app bundle, parses it using the GPX

[13] https://github.com/Kotlin/multiplatform-library-template
[14] https://touchlab.co/kmmbridge

parser, and returns the route as a list of coordinates that can be displayed on the map.

Compose Multiplatform Resources allow sharing images, strings, fonts, or raw files.

Since we are not working with data from remote sources, we need to read resources across platforms as streams or binary files. Compose Multiplatform (CMP) Resources[15] allow integrating resources such as images, strings, fonts, or raw files into a shared KMP library. However, the library is intended to be used with the multiplatform UI framework Compose Multiplatform.

CMP Resources did not satisfy our requirements.

Initially, we tried using CMP Resources to read the GPX file. While it was possible to read JSON files by path using `Res.readBytes`, we could not get a list of all files in a folder using CMP Resources.

A shared `FileManager` enables uniform access of files in content bundle across platforms.

Since it was important to us to be able to read the list of files in a specific folder, we decided to leave the content in the respective native app bundles[16]. We instead implemented a shared `FileManager` as a dedicated abstraction. It enables us to load any files from the app resources via a uniform interface using a relative path (e.g., `pois/index.json`) and list the contents of a directory.

The expect/actual syntax enables defining shared interfaces in `commonMain` with platform-specific native implementations.

To write platform-specific implementations, we used the expect/actual syntax. This syntax allows specifying the expected interface in the `commonMain` source set, similar to an interface in the context of object-oriented languages. We implement this behavior in the platform-specific source sets using native APIs (e.g., Foundation's `FileManager` on iOS and `Context.assets` on Android).

`PathResolver` enables cross-platform resource access by resolving platform-specific paths.

Additionally, we wrote a `PathResolver` based on the same principle to use relative paths across platforms to read the resources. The `PathResolver` returns the platform-specific URL or path of the file by using the `resourceURL` of a `Bundle` on iOS and uses the `Context` for Android.

[15] https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-resources.html

[16] We later implemented a command line command that copies the app content to the correct locations to improve developer experience.

The unified interface ensures reliable file loading, using platform-specific implementations even in common code.

This unified interface uses platform-specific implementations to reliably load files on upper layers. The platform-specific implementation is also used to reference expected classes in common code.

Utilizing the newly implemented FileManager, we can now read the GPX file from the app's resources in the `LocalRouteService`, convert it into a GPX object with the `GPXParser` and then return the route as a list of coordinates.

4.3.3 Architecture Considerations & Dependency Injection

We identified problems and designed a new app architecture.

After our successful initial tests with KMP, we needed to identify the current problems and design a suitable app architecture.

The old code used a general loader on iOS and Android.

Figure 4.4 shows the old architecture of FLApp's iOS and Android implementation business layer. Both implementations use a general-purpose class for loading the content.

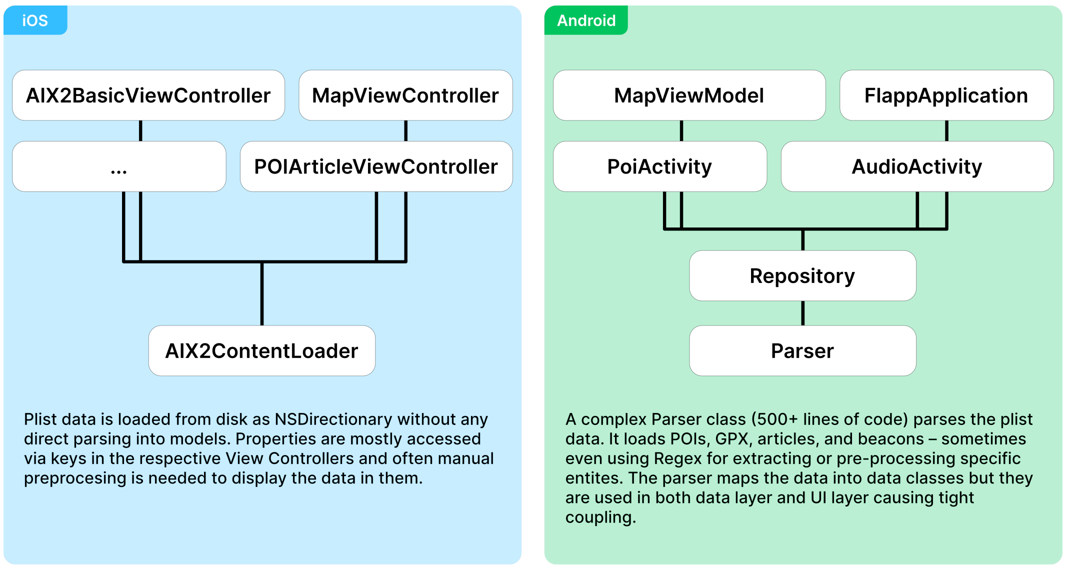

iOS
AIX2BasicViewController
MapViewController
...
POIArticleViewController
AIX2ContentLoader
Plist data is loaded from disk as NSDirectionary without any direct parsing into models. Properties are mostly accessed via keys in the respective View Controllers and often manual preprocesing is needed to display the data in them.
Android
MapViewModel
FlappApplication
PoiActivity
AudioActivity
Repository
Parser
A complex Parser class (500+ lines of code) parses the plist data. It loads POIs, GPX, articles, and beacons – sometimes even using Regex for extracting or pre-processing specific entites. The parser maps the data into data classes but they are used in both data layer and UI layer causing tight coupling.

Figure 4.4: In the old implementations, loading the content (e.g., POIs, articles, GPX, etc.) from disk involved a centralized loader/parser class, which the UI used directly.

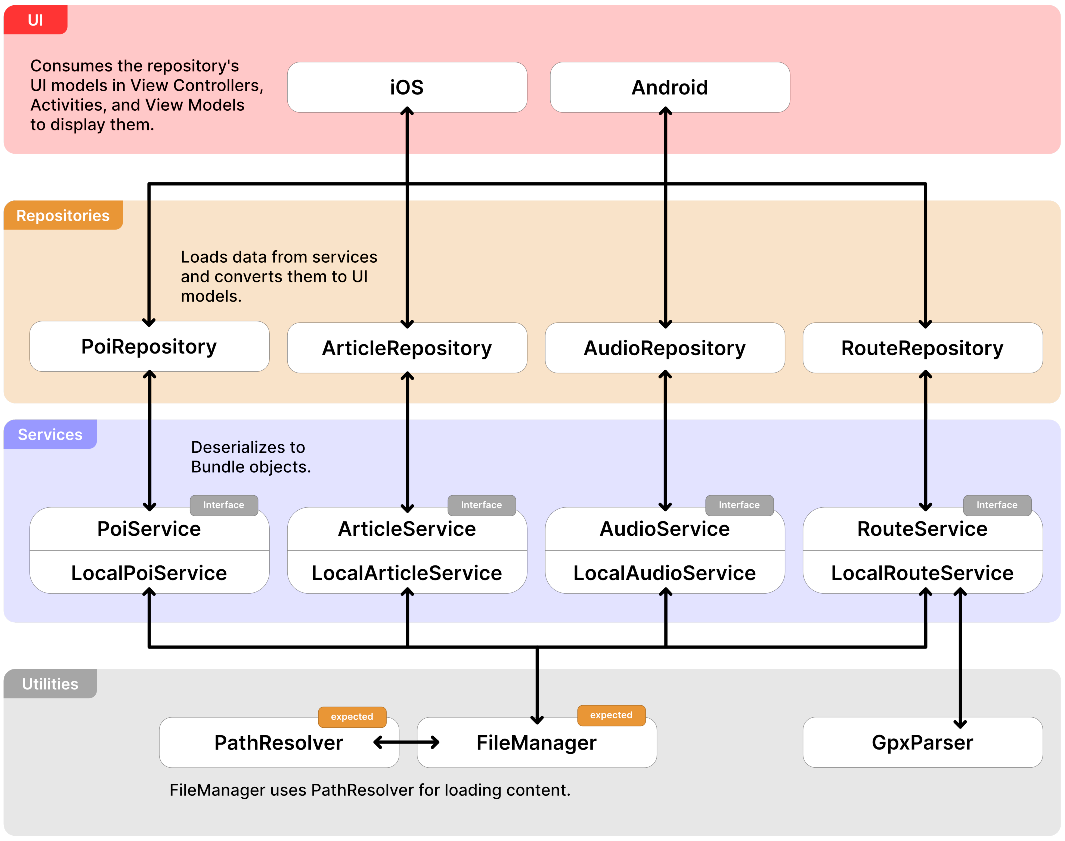


Figure 4.5: We planned a layered architecture consisting of a service layer loading and parsing the data via the FileManager. The repository layer consumes data from the services and maps the deserialized data to decoupled UI objects. Native UI classes like View Controllers, Activities, and View Models use these objects. Expected classes have actual platform-specific implementations.

The iOS implementation loads plist data from disk as `NSDirectionary` without direct parsing into data classes. The View Controllers mainly access the properties via string keys. Often, manual pre-processing is used in the View Controllers to display the data correctly.

iOS loaded plist data as NSDictionaries and accessed properties via string keys in View Controllers.

A complex and hard-to-maintain `Parser` class (500+ lines of code) is used to parse the plist data on Android. It loads POIs, GPX, articles, and beacons, sometimes using Regex to extract or pre-process specific entities. The parser converts the data into data classes used in both the data and UI layers, causing tight coupling.

Android used a complex Parser class to convert plist data to data classes.

The current implementations violate SRP, single source of truth, and have tight coupling between UI and data.

The Single Responsibility Principle (SRP) [Martin, 2017] is violated as the parsers load data for many places. Supposedly, both platform implementations are considered as one project in the context of cross-platform migration. In that case, two data sources violate the single source of truth principle [Long, 2021]. Furthermore, the UI's direct access to raw data brings tight coupling, which should generally be avoided as it makes changing the data format and testing difficult [Martin, 2008].

We planned a layered architecture with services and repositories to decouple data and UI.

To mitigate these problems, we planned the architecture visualized in Figure 4.5. We introduce a layered approach with services and repositories for decoupling the data and UI layer. Each service only loads specific data and has a single responsibility. In the repository layer, services load the data and map it into UI objects, which are then used to present the user interface.

Using separate data classes for data and UI layers enforces information hiding, allowing UI optimization.

Using distinct data classes for data and UI layers implements the principle of information hiding [Martin, 2017]. This architectural separation ensures that UI components are isolated from the specifics of the data layer's format. It also allows independent optimization of UI-centric data structures for optimal presentation across screens without altering the foundational data model.

DI was adopted to improve maintainability and decoupling of business logic.

Since we are now incorporating more classes with multiple levels of abstraction into the project in the business layer, it is practical to adopt dependency injection (DI) as a design principle.

Excursus: *Dependency Injection*

DEPENDENCY INJECTION:
In *Dependency Injection Principles, Practices, and Patterns* Seemann and van Deursen [2019] define dependency injection as "a set of software design principles and patterns enabling developing loosely coupled code."

DI helps us write more maintainable and loosely coupled code.

In our context, DI mainly involves injecting all dependencies that a class requires into it at the time of its creation in a uniform way. Loosely coupled code makes implementations more maintainable [Seemann and van Deursen, 2019]. It simplifies testing, exchanging implementations, and mocking for previews (e.g., with SwiftUI or Jetpack Compose).

To improve the developer experience, we decided to use a popular DI library. Koin[17] supports KMP and thus allows a standardized approach for DI on iOS and Android.

We selected Koin as a DI framework for FLApp.

For this purpose, we added Koin modules (not to be confused with Gradle modules) to the common iOS and Android source sets. Koin modules use a *Domain Specific Language* (DSL) to specify how objects of a particular type should be resolved when being "requested" (also called *injected*) via the Koin DI framework (Listing 4.2, Listing B.6).

Koin modules define how to resolve and inject dependencies.

```
val sharedModule = module {
    single<GpxParser>(createdAtStart = true) {
        GpxParser()
    }
    single<RouteService>(createdAtStart = true) {
        LocalRouteService(gpxParser = get())
    }
}
```

Listing 4.2: Definition of a `RouteService` being resolved by an instance of a `LocalRouteService` in a Koin module.

We register platform-specific instantiations for specific classes in the iOS and Android Koin modules. This is especially useful if platform-specific implementations need access to other platform-specific objects. For example, many Android-specific features (such as loading files) require the Android `Context`. Koin makes managing and instantiating different implementations based on a target platform or environment much more manageable (Listing 4.3).

Platform-specific Koin modules register platform-specific dependencies.

[17] https://insert-koin.io

```kotlin
val androidModule = module {
    single<FileManager>() {
        return@single DefaultFileManager(
            context = androidContext()
        )
    }
}
```

Listing 4.3: Platform specific dependency injection using Koin.

Koin is initialized in each app to use shared and platform-specific modules.

Koin then only needs to be initialized in the respective apps so that the Koin modules `sharedModule` and `androidModule` or `iosModule` are used and the Android `Context` is bound (Listing 4.4).

```kotlin
startKoin {
    androidContext(this@FlappApplication)
    androidLogger()
    modules(sharedModule, androidModule)
}
```

Listing 4.4: Koin needs to be initialized on both platforms, e.g., in `onCreate` of `FlappApplication`.

4.3.4 Moving to a new Data Format

Now that we have successfully built our first business logic feature to KMP, we want to continue unifying other parts of the business logic.

Using plist files was not practical outside of Apple's ecosystem.

A significant challenge in the original implementations was the reliance on the property list format (`.plist`). The initial version of FLApp was developed as an iOS-only project, historically using the plist format for content storage. With the introduction of the Android version, the `plist` format had to be adopted to maintain compatibility. However, this choice proved less than ideal, as plist files are uncommon outside Apple's ecosystem and lack portability across platforms[18].

[18] https://mediawiki.gnustep.org/index.php/Property_Lists

Recognizing this limitation, we decided to transition to a JSON-based format. However, the exact specification of our new data format was still being determined at the time of writing. For that reason – and to allow future maintenance – we designed our implementation with a level of abstraction. This allows for easy adaptability and seamless transitions to alternative data formats if needed. We designed a JSON-based format for testing purposes. However, we will not go into detail here as this is not the main subject of our work.

We switched to a JSON-based format and designed our implementation to support changing formats.

To achieve this level of abstraction, we have separated the entities from the data layer and entities for the UI layer from each other and connected them using a bidirectional mapper. This way, the UI does not automatically have to be adapted when the underlying data layer changes.

We separated data layer models from UI layer models.

As a convention, we use the prefix `Bundle` and the postfix `Ui` to distinguish objects belonging to these two layers. Additionally, our goal was to use individual `Ui` entities for each screen and keep them as lightweight as possible, containing only the data needed for that screen.

We designed lightweight UI entities for each screen to reduce complexity.

We have also incorporated this paradigm for our provisional JSON-based data format Figure 4.6. The list of POIs `pois/index.json` only contains the information required to display the POIs on the map and the sidebar. Unlike in the previous data format, each POI now also has a unique ID, which is used to load the detailed information from `pois/{poiId}.json`. Therefore, a detailed POI request also returns the article teasers, information about the AR content, the audio ID, and a quote.

The new data format uses index files and unique IDs to load detailed information for each item.

As we explained in subsection 4.3.3, we have added additional services (`LocalPoiService`, `LocalAudioService`, `LocalArticleService`, and `LocalBeaconService`) that load the required data for the user interface in a REST-like fashion (index and show). Even though we deliver the content entirely locally and offline in the app, it was important to us to use a popular pattern here.

We use local REST-like services for UI data, enabling future remote API support.

Each service uses the before implemented `FileManager` to load the desired resources from the disk, parses them, and

Services load resources using FileManager and deserialize them.

```
content
├── articles
│   └── {articleId}.json
├── audio
│   └── {audioId}.json
├── pois
│   ├── index.json
│   └── {poiId}.json
├── media
│   ├── audio
│   ├── images
│   └── videos
├── beacons
│   └── index.json
└── track.gpx
```

Figure 4.6: We used a more granular structure for the new data model.

then passes them on as data classes. Bundle objects are deserialized using `kotlinx.serialization` into corresponding data classes and then returned by the service (Listing B.7).

All services are registered in Koin DI.

All services are integrated into the Koin dependency injection, allowing them to be quickly injected at the required locations in the shared code and native targets.

We were able to enforce the Single Responsibility Principle, remove the tight coupling, and establish a single source of truth across platforms. At some points, we also had to change the data handling in the user interface. Especially on iOS, we had to fix and improve the article screen to fit the new data model.

4.3.5 Data Migration Using KMP

KMP was used to automate conversion to new JSON data format.

When working on a new JSON data format, we first manually created files to find a suitable one. However, we were concerned it would be tedious to do it manually for all app content. We then realized that we could also use

Kotlin Multiplatform for that task. By adding an additional platform-agnostic module to our application, we could, in theory, use our already written shared code for the definitions, map the old legacy format to the new data format, and save it to disk on the developer's machine.

We started implementing this idea by adding another KMP module called `script` to our application. We aimed to use the original Android parser to parse the old content and then map it to the new data format. Unfortunately, the parser used a Java-only dependency to parse the plist format. This dependency issue meant we would only support a Java Virtual Machine (JVM) target to run the code in our KMP script module. However, as this script module was only meant to run on the developer's machine, that imposed no significant problems.

The script module needed a JVM target to utilize a Java parser dependency.

The only problem this caused was the need to define a JVM target in our shared KMP module because our script module requires the shared module as a dependency. We therefore needed to implement the expected platform-specific implementations for the `PathResolver`, `FileManager`, and `Preferences`. It is needed as Kotlin fails to compile for a specific target if expected classes are missing an actual implementation. We just implemented those classes with mocked behavior for the compilation to pass, as we did not need them in our script module.

Mocked JVM classes were implemented to enable compilation in the script module.

Then, we moved all legacy Android source files needed for the original `Parser` to the module. After minor adjustments, we successfully used the old Android parser implementation, mapped the parsed objects to the new data format, and saved them to disk. Running `./gradlew run -q --args="convert"` converts the old content folder into the new format and `./gradlew run -q --args="copy"` copies them to the correct locations in the target iOS and Android apps.

We used Legacy Android code in the KMP module to convert legacy data to the new format.

We used Clikt[19] for building and especially parsing command line commands. The package supports KMP and works with JVM, Linux, Windows, and macOS.

Clikt was used for command line parsing in the script.

[19] https://ajalt.github.io/clikt

We implemented automated structural assessment for detecting differences between languages.

In addition to transforming the original data model to the new format, we also implemented the capability to find structural differences between German and English content, enabling automated assessment of content structure moving forward. Implementing more developer Quality of Life (QoL) improvements like this is suitable for future work.

Chapter 5

Evaluation

In the evaluation, we discuss our development experience with KMP as a cross-platform technology for FLApp, the effectiveness of our migration strategy, and the fulfillment of our previous goal.

No user study as a performance degradation was very unlikely.

We did not conduct a user study because we continued to use our native user interface. Therefore, users will be unable to detect any visual differences except responsiveness. However, we knew from our research on KMP that we could expect a negligible performance degradation for the capabilities used in our app on Android and especially iOS [Skantz, 2023].

Incremental migration was effective; migrating to Swift first was useful.

The incremental migration was the right choice. This approach proved effective, as incremental changes allowed for iterative testing and step-by-step reduction of inconsistent and duplicated code. Also, the decision to migrate to Swift first proved advantageous, allowing us to leverage the similarities between Swift and Kotlin more effectively and directly.

Keeping native UI was right; CMP allows future UI changes.

Furthermore, we are also convinced that keeping the native UI implementations was a good choice (Goal 1). Reimplementing the entire user interface in cross-platform technology would not have been very valuable as, despite content updates, there are currently no plans for this app to get any new features. However, through Compose Multiplatform, we still have the flexibility to incrementally replace or implement new screens across multiple platforms.

Separation of concerns and single responsibility achieved.

We successfully achieved our goal of separation of concerns and single responsibility (Goal 2). We developed reusable, maintainable, and testable components by adhering to the single responsibility principle and creating well-defined abstractions (Goal 3).

Cross-platform code eliminated inconsistencies; tests enhanced stability.

The introduction of cross-platform code in the business layer eliminated inconsistencies and, therefore, yielded a more predictable and uniform experience for users and developers. The introduction of tests further enhanced confidence and stability, guiding future developers working on the project.

Decoupling data from UI provides flexibility.

We gained the flexibility to modify the underlying data format without impacting UI code by decoupling presentation and data models. This separation ensures adaptability for future changes to the data model.

KMP's expect/actual mechanism was very convenient for deep integration.

From a developer perspective, we liked the expect/actual mechanism, as it felt akin to writing object-oriented code with interfaces and implementations. This approach allowed us to concentrate on software architecture rather than fighting against the framework. This level of abstraction and seamless integration of native platform implementations into common code felt unique to KMP and, in our opinion, would have been difficult – if not impossible – to achieve with other frameworks like React Native or Flutter.

Unified implementation solved inconsistencies and eliminated duplication.

While we do not have concrete metrics on our developer productivity, the perceived speed and level of improvement were high. We achieved a unified implementation from the first line of multiplatform code, effectively resolving existing inconsistencies. Shared classes and data structures eliminated duplication, resulting in cleaner, more maintainable, and cohesive logic. Additionally, integrating shared modules into script-related functionalities demonstrated the advantages of platform-independent runtime requirements, further reinforcing the value of a shared codebase.

Although we have had much prior experience with iOS, Swift, Android, and Kotlin, we generally found the available tools to provide a good development experience. Despite the general understanding of the different source sets and implementing shared and platform-specific code, we found that native developers will feel right at home as they have most of their default tools and do not need to learn a substantially different language or paradigm. This also holds for the UI framework CMP, built on top of Android's Jetpack Compose. However, this will be different for developers with a web background, as a fair amount of platform-specific knowledge is needed to use KMP productively. They would probably have a better start with React Native.

KMP provides a good experience for native developers.

When starting with Kotlin Multiplatform (KMP), we found setting up Gradle for Multiplatform projects in an existing codebase challenging, particularly for developers with limited experience with the build tool. To ease the learning curve, we recommend using the KMP project wizard as a reference for configuration. Fortunately, JetBrains is actively addressing these challenges by developing the Amper build tool, which is designed to simplify Multiplatform project setup and management.

Gradle setup was challenging, but JetBrains will improve this.

Although we used relatively few Kotlin Multiplatform (KMP) libraries overall – primarily because we did not need them – one library that stood out was the Koin dependency injection framework, which significantly benefited our project. The primary challenge was integrating with iOS, where resolving dependencies required additional syntactic sugar to be practical.

Koin dependency injection was beneficial but had iOS challenges.

The reliance on Objective-C as a bridging layer for the interoperability between Kotlin and Swift presented challenges. Especially for developers not familiar with Objective-C's data structure, this is not straightforward. Managing multi-threaded asynchronous code proved technically challenging, as the Objective-C bridge compounded differences in thread handling between Kotlin and Swift. We hope the announced direct Kotlin-to-Swift export will improve this and the general interoperability.

Kotlin/Swift interoperability through Objective-C had challenges.

Overall, we are very satisfied with Kotlin Multiplatform (KMP) and our migration experience. All initial goals were successfully fulfilled, effectively addressing the underlying issues.

Chapter 6

Summary and Future Work

6.1 Summary and Contributions

The main contribution of this work is the detailed description of a native-to-cross-platform migration of a real app using Kotlin Multiplatform: We overviewed current research, found a gap in scientific research on cross-platform migrations and especially up-to-date comparable literature on cross-platform technologies, and presented some case studies. Furthermore, we explained our rationale, goals, and requirements for the cross-platform migration. We documented our decision-making and comparison process for selecting a suitable cross-platform framework for FLApp. Moreover, we described our migration strategy and technical details. Finally, we evaluated our migration success from a developer perspective, especially KMP as a cross-platform technology.

We contributed a detailed native-to-KMP migration.

6.2 Future Work

In the context of future work, we propose to focus on three primary areas: scientific research on cross-platform frameworks (CPFs) and migrations, further investigation of Kotlin Multiplatform (KMP) as a technology, and the continued development of FLApp.

More up-to-date and comparable research on CPFs is needed.

First, updated and comparable research on cross-platform technologies remains a significant need. Many existing studies present outdated frameworks, and as noted by Rieger and Majchrzak [2019], there needs to be more unified, scientifically comparable research in this area. Establishing a standardized, quantifiable foundation for evaluating cross-platform frameworks through consistent testing and empirical studies could support the creation of decision frameworks. This research would enable more informed selections of appropriate technologies. However, it is also important to emphasize that the project's specific requirements should always drive the choice of technology.

Objective-C bridging and Gradle setup still pose challenges.

Currently, when integrating Kotlin code into the iOS ecosystem, developers can still feel the limitations of the Objective-C bridging layer. This integration adds complexity and negatively impacts developer experience, especially regarding asynchronous code. Also, the Gradle setup can be difficult for Gradle beginners.

JetBrains is addressing these limitations with a direct Swift export and Amper.

JetBrains is actively addressing the well-recognized development challenges. As mentioned, the company is working on a direct Kotlin-to-Swift export feature and a new KMP build tool called Amper, which aims to streamline the development process.

Further development could extend KMP to the presentation layer.

Further development of FLApp could involve expanding the use of KMP into the presentation layer or adopting Compose Multiplatform for parts of the user interface. Steps like these would result in an even greater unification of the codebase.

Based on the migration module developed in this work, extending it with Compose Multiplatform would also be possible. Adopting CMP could facilitate the creation of small utilities that allow for further editing or adding content for FLApp during development and maintenance.

A Compose Multiplatform utility for editing app content for desktop or web could be created.

Appendix A

Supporting Figures

This page is intentionally left blank.

Perspective	Criterion
App perspective	Hardware Access (A1)
	Platform Functionality (A2)
	Connected Devices (A3)
	Input Heterogeneity (A4)
	Output Heterogeneity (A5)
	Application Lifecycle (A6)
	System Integration (A7)
	Security (A8)
	Robustness (A9)
	Degree of Mobility (A10)
Infrastructure perspective	License (I1)
	Target Platforms (I2)
	Development Platforms (I3)
	Distribution Channels (I4)
	Monetisation (I5)
	Internationalisation (I6)
	Long-term Feasibility (I7)
Development perspective	Development Environment (D1)
	Preparation Time (D2)
	Scalability (D3)
	Development Process Fit (D4)
	UI Design (D5)
	Testing (D6)
	Continuous Delivery (D7)
	Configuration Management (D8)
	Maintainability (D9)
	Extensibility (D10)
	Custom Code Integration (D11)
	Pace of Development (D12)
Usage perspective	Look and Feel (U1)
	Performance (U2)
	Usage Patterns (U3)
	User Authentication (U4)

Table A.1: Criteria presented by Rieger and Majchrzak [2019] in their decision framework for selecting a suitable cross-platform framework.

Appendix B

Code Samples

```kotlin
// --------------- Shared source set ---------------

expect fun randomUUID(): String

// --------------- Android source set --------------

import java.util.*

actual fun randomUUID(): String = UUID.randomUUID().toString()

// --------------- iOS source set ------------------

import platform.Foundation.NSURL

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

Listing B.1: Kotlin uses the expect/acutal syntax, which is similar to interfaces and implementations. In the shared code, we define an expected class with a number of methods and properties. In the platform-specific source sets, we can provide actual implementations for these declarations also using platform-specific APIs. In this example, only a function returning a random uuid is expected. The Android and iOS source sets use platform APIs to return a random uuid string.

```kotlin
rootProject.name = "FLApp"
enableFeaturePreview("TYPESAFE_PROJECT_ACCESSORS")

pluginManagement {
    repositories {
        // ...
    }
}
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version "
        0.8.0"
}

dependencyResolutionManagement {
    repositories {
        google {
            // ...
        }
        mavenCentral()
    }
}

include(":shared")
include(":androidApp")

include(":sceneform")
project(":sceneform").projectDir = File("sceneformsrc/sceneform")

include(":sceneformux")
project(":sceneformux").projectDir = File("sceneformux/ux")

include(":script")
```

Listing B.2: The Gradle settings at `settings.gradle.kts` contain additional registration of the shared and script module. We omit the plugin and dependency resolution management for readability reasons.

```
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi
import org.jetbrains.kotlin.gradle.dsl.JvmTarget

plugins {
    alias(libs.plugins.kotlinMultiplatform)
    alias(libs.plugins.androidLibrary)
    alias(libs.plugins.kotlinxSerialization)
    alias(libs.plugins.ksp)
    alias(libs.plugins.kmpNativeCoroutines)
    alias(libs.plugins.jetbrainsCompose)
    alias(libs.plugins.compose.compiler)
}

kotlin {
    androidTarget {
        @OptIn(ExperimentalKotlinGradlePluginApi::class)
        compilerOptions {
            jvmTarget.set(JvmTarget.JVM_11)
        }
    }

    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach {
        it.binaries.framework {
            baseName = "Shared"
            isStatic = true
        }
    }

    jvm()

    sourceSets {
        androidMain.dependencies {
            implementation(libs.androidx.preference.ktx)
            implementation(libs.koin.android)
        }
        iosMain.dependencies {
            // No other dependencies
        }
        commonMain.dependencies {

            // Compose Multiplatform
            implementation(compose.runtime)
            implementation(compose.foundation)
            implementation(compose.material)
            implementation(compose.ui)
            implementation(compose.components.resources)
            implementation(compose.components.uiToolingPreview)
            implementation(libs.androidx.lifecycle.viewmodel)
            implementation(libs.androidx.lifecycle.viewmodel.compose)
            implementation(libs.androidx.lifecycle.runtime.compose)
            implementation(libs.kermit)

            // Koin
            implementation(project.dependencies.platform(libs.koin.bom)
                )
            implementation(libs.koin.core)
            implementation(libs.koin.compose)
            implementation(libs.koin.compose.viewmodel)

            api(libs.kmm.viewmodel)
            implementation(libs.xmlutil.core)
            implementation(libs.xmlutil.serialization)
            implementation(libs.kotlinx.serialization.json)
            implementation(libs.kotlinx.io.core)
            implementation(libs.dd.plist)
            implementation(compose.components.resources)
            implementation(compose.runtime)
        }

        commonTest.dependencies {
            implementation(kotlin("test"))
        }

        // Required by KMM-ViewModel
        all {
            languageSettings.optIn("kotlinx.cinterop.
                ExperimentalForeignApi")
            languageSettings.optIn("kotlin.experimental.
                ExperimentalObjCName")
        }
    }
}
```

```

android {
    namespace = "aachen.rwth.de.flapp.shared"
    compileSdk = libs.versions.compileSdk.get().toInt()
    defaultConfig {
        minSdk = libs.versions.minSdk.get().toInt()
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_11
        targetCompatibility = JavaVersion.VERSION_11
    }
}
```

Listing B.3: The Gradle build file of the shared module at `shared/build.gradle.kts`. Interesting in this context are the targets and source sets. `androidTarget`, `iosX64`, `iosArm64`, `iosSimulatorArm64`, and `jvm` define the targets of the module. In the source sets, we specify the dependencies of the source sets and testing source sets.

```kotlin
// --------------- Shared source set ---------------

expect class FileManager {

    fun list(path: String): Array<String>

    fun load(path: String): String

}

// --------------- Android source set --------------

package aachen.rwth.de.flapp.shared.utils.io

import kotlinx.cinterop.BetaInteropApi
import platform.Foundation.NSBundle
import platform.Foundation.NSData
import platform.Foundation.NSFileManager
import platform.Foundation.NSString
import platform.Foundation.NSUTF8StringEncoding
import platform.Foundation.create
import platform.Foundation.dataUsingEncoding

@OptIn(BetaInteropApi::class)
fun String.data(): NSData? =
    NSString.create(string = this).dataUsingEncoding(NSUTF8StringEncoding)

@OptIn(BetaInteropApi::class)
fun NSData.string(): String? =
    NSString.create(data = this, encoding = NSUTF8StringEncoding)?.toString()

actual class FileManager {

    private val baseUrl = NSBundle.mainBundle.resourceURL!!
    private val dataResourceUrl = baseUrl.URLByAppendingPathComponent("Data")!!
    private val fileManager = NSFileManager.defaultManager

    override fun load(path: String): String {
        val data = fileManager.contentsAtPath(path) ?: throw Exception("File not
            found")

        data.string()?.let {
            return it
        } ?: throw Exception("Data could not be converted to string")

    }

    override fun list(path: String): Array<String> {
        val url = dataResourceUrl.URLByAppendingPathComponent(path) ?: throw
            Exception("Directory not found")
        val contents = fileManager.contentsOfDirectoryAtPath(url.path ?: throw
            Exception("Directory not found"), null) ?: throw Exception("Directory
            not found")

        return contents.map { it.toString() }.toTypedArray()
    }

}

// --------------- iOS source set ------------------

package aachen.rwth.de.flapp.shared.utils.io

import aachen.rwth.de.flapp.shared.utils.io.FileManager
import android.content.Context
import kotlinx.serialization.json.Json
import java.io.IOException

actual class FileManager(context: Context) {

    private val assets = context.assets

    @Throws(IOException::class)
    override fun list(path: String): Array<String> {
        return assets.list(path) ?: emptyArray()
    }

    override fun load(path: String): String {
        return assets.open(path).bufferedReader().use { it.readText() }
    }

}
```

Listing B.4: This version of the FileManager is simplified, but contains all neccessary functionality. The expected class specifies the *interface* and both actual implementations use platform-specific APIs.

```kotlin
package aachen.rwth.de.flapp.shared

// Omitted imports

/**
 * Shared module that provides all shared dependencies.
 */
val sharedModule = module {

    // ----------- Utils -----------

    single<UserInterfaceStateRepository>(createdAtStart = true) {
        UserInterfaceStateRepository(
            preferences = get(),
            arUtils = get()
        )
    }

    single<Json> {
        return@single Json {
            ignoreUnknownKeys = true
            explicitNulls = false
            serializersModule = SerializersModule {
                polymorphic(BundleContentBase::class) {
                    subclass(BundleContentText::class)
                    subclass(BundleContentImage::class)
                    subclass(BundleContentArLink::class)
                    subclass(BundleContentTitle::class)
                    subclass(BundleContentVideo::class)
                    subclass(BundleContentCredits::class)
                    subclass(BundleContentSubheading::class)
                }
            }
        }
    }

    // ----------- Route -----------
    single<GpxParser>(createdAtStart = true) {
        GpxParser()
    }
    single<RouteService>(createdAtStart = true) {
        LocalRouteService(
            gpxParser = get(),
            fileManager = get()
        )
    }
    single<RouteRepository>(createdAtStart = true) {
        RouteRepository(
            routeService = get()
        )
    }

    // ----------- Beacon -----------

    single<BeaconService>(createdAtStart = true) {
        LocalBeaconService(
            fileManager = get(),
            json = get()
        )
    }

    // ----------- Poi -----------

    single<PoiService>(createdAtStart = true) {
        LocalPoiService(fileManager = get(), json = get())
    }
    single<PoiRepository>(createdAtStart = true) {
        return@single PoiRepository(
            poiService = get()
        )
    }

    // ----------- Article -----------

    single<ArticleService>(createdAtStart = true) {
        LocalArticleService(fileManager = get(), json = get())
    }
    single<ArticleRepository> {
        return@single ArticleRepository(
            articleService = get()
        )
    }

    single<AudioService>(createdAtStart = true) {
        LocalAudioService(fileManager = get(), json = get())
    }

    viewModelOf(::DebugViewModel)

}
```

Listing B.5: This Koin module defines the dependencies shared across the application.

```
package aachen.rwth.de.flapp

import aachen.rwth.de.flapp.shared.utils.io.DefaultFileManager
import aachen.rwth.de.flapp.shared.utils.preferences.Preferences
import aachen.rwth.de.flapp.shared.utils.io.FileManager
import org.koin.android.ext.koin.androidContext
import org.koin.dsl.module

val androidModule = module {
    single<Preferences>(createdAtStart = true) {
        Preferences(context = androidContext())
    }
    single<FileManager>() {
        return@single DefaultFileManager(context = androidContext())
    }
}
```

Listing B.6: This Koin module defines the Android-specific dependencies registered in the application.

```kotlin
package aachen.rwth.de.flapp.shared.pois

import aachen.rwth.de.flapp.shared.utils.io.PathResolver
import aachen.rwth.de.flapp.shared.utils.io.FileManager
import kotlinx.io.IOException
import kotlinx.serialization.json.Json

class LocalPoiService(
    private val fileManager: FileManager,
    private val json: Json
) {

    fun getPois(): List<BundlePoi> {

        val data = fileManager.load(PathResolver.resolve("pois/index.json"))
        val index = json.decodeFromString<BundlePoisIndex>(data)

        return index.data

    }

    fun getPoiDetail(poiId: Int): BundlePoiDetail? {
        try {
            val stream = fileManager.load(PathResolver.resolve("pois/${poiId}.json"))
            return json.decodeFromString(stream)
        } catch (e: IOException) {
            e.printStackTrace()
        }
        return null
    }

}
```

Listing B.7: This listing shows an example service having the single responsibility to use the FileManager for loading the data and then parsing it into Bundle objects that are then returned.

Bibliography

[1] Suyesh Amatya and Arianit Kurti. Cross-Platform Mobile Development: Challenges and Opportunities. In Vladimir Trajkovik and Misev Anastas, editors, *ICT Innovations 2013*, pages 219–229, Heidelberg, 2014. Springer International Publishing.

[2] Esteban Angulo and Xavier Ferre. A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX. In *Proceedings of the XV International Conference on Human Computer Interaction*, Interacción '14, New York, NY, USA, 2014. Association for Computing Machinery. `doi.org/10.1145/2662253.2662280`.

[3] Oscar Axelsson and Fredrik Carlström. Evaluation Targeting React Native in Comparison to Native Mobile Development. Master's thesis, Lund University, 2016. URL `https://api.semanticscholar.org/CorpusID:114710275`.

[4] Heiko Behrens. Cross-Platform App Development for iPhone, Android & Co.— A Comparison. In *MobileTechCon*, 2010.

[5] Fu Cheng. *Platform Integration*, pages 441–471. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4982-6. `doi.org/10.1007/978-1-4842-4982-6_12`.

[6] Yoonsik Cheon and Carlos Chavez. Converting Android Native Apps to Flutter Cross-Platform Apps. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1898–1904, 2021. `doi.org/10.1109/CSCI54926.2021.00355`.

[7] Luis Corral, Andrea Janes, and Tadas Remencius. Potential Advantages and Disadvantages of Multiplatform Development Frameworks - A Vision on Mobile Environments. *Procedia Computer Science*, 10:1202–1207, 2012. `doi.org/10.1016/j.procs.2012.06.173`. ANT 2012 and MobiWIS 2012.

[8] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854, 2017. `doi.org/10.1145/3126594.3126651`.

[9] B. Eisenman. *Learning React Native: Building Native Mobile Apps with JavaScript*. O'Reilly Media, 2015. ISBN 9781491929070.

[10] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef, and Ayman M. Wahba. Taxonomy of Cross-Platform Mobile Applications Development Approaches. *Ain Shams Engineering Journal*, 8(2):163–190, 2017. `doi.org/10.1016/j.asej.2015.08.004`.

[11] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, and Emerson Murphy-Hill. Advantages and disadvantages of a monolithic repository: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 225234, New York, NY, USA, 2018. Association for Computing Machinery. `doi.org/10.1145/3183519.3183550`.

[12] Mohamed Karim Khachouch, Ayoub Korchi, Younes Lakhrissi, and Anis Moumen. Framework Choice Criteria for Mobile Application Development. In *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–5, June 2020. `doi.org/10.1109/ICECCE49384.2020.9179434`.

[13] Mohamed Lachgar, Mohamed Lachgar, Mohamed Hanine, Mohamed Hanine, Hanane Benouda, Hanane Benouda, Younes Ommane, and Younes Ommane. Decision framework for cross-platform mobile development frameworks using an integrated multi-criteria decision-making methodology. *International Journal of Mobile Computing and Multimedia Communications*, 2022. `doi.org/10.4018/ijmcmc.297928`.

[14] Tom Long. *Good Code, Bad Code: Think like a software engineer*. Manning, 2021.

[15] Tim A. Majchrzak, Tim A. Majchrzak, Andreas Biørn-Hansen, Andreas Biørn-Hansen, Tor Morten Grønli, and Tor-Morten Grønli. Progressive Web Apps: the Definite Approach to Cross-Platform Development? *Hawaii International Conference on System Sciences*, 2018. `doi.org/10.24251/hicss.2018.718`.

[16] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882.

[17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall Press, USA, 1st edition, 2017. ISBN 0134494164.

[18] Piotr Nawrocki, Krzysztof Wrona, Mateusz Marczak, and Bartlomiej Sniezynski. A Comparison of Native and Cross-Platform Frameworks for Mobile Applications. *Computer*, 54(3):18–27, 2021. `doi.org/10.1109/MC.2020.2983893`.

[19] Robin Nunkesser. Beyond web/native/hybrid: a new taxonomy for mobile app development. In *Proceedings of the 5th International Conference on Mobile*

Software Engineering and Systems, MOBILESoft '18, page 214218, New York, NY, USA, 2018. Association for Computing Machinery. doi.org/10.1145/3197231.3197260.

[20] Wellington Oliveira, Bernardo Moraes, Fernando Castor, and João Paulo Fernandes. Analyzing the Resource Usage Overhead of Mobile App Development Frameworks. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, pages 152–161, New York, NY, USA, 2023. Association for Computing Machinery. doi.org/10.1145/3593434.3593487.

[21] Akshat Paul and Abhishek Nalwaya. *Native Bridging in React Native*, pages 165–186. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4454-8. doi.org/10.1007/978-1-4842-4454-8_7.

[22] Rap Payne. *Developing in Flutter*, pages 9–27. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-5181-2. doi.org/10.1007/978-1-4842-5181-2_2.

[23] Rap Payne. *Everything Is Widgets*, pages 31–46. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-5181-2. doi.org/10.1007/978-1-4842-5181-2_3.

[24] Carlos Manso Pinto and Carlos Coutinho. From Native to Cross-platform Hybrid Development. In *2018 International Conference on Intelligent Systems (IS)*, pages 669–676, 2018. doi.org/10.1109/IS.2018.8710545.

[25] C.P Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, 2012. doi.org/10.1109/INDCON.2012.6420693.

[26] Christoph Rieger and Tim A. Majchrzak. Towards the definitive evaluation framework for cross-platform app development approaches. *Journal of Systems and Software*, 153:175–199, 2019. doi.org/10.1016/j.jss.2019.04.001.

[27] Dominik Schultes. SequalsKA Bidirectional Swift-Kotlin-Transpiler. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 73–83, 2021. doi.org/10.1109/MobileSoft52590.2021.00017.

[28] M. Seemann and S. van Deursen. *Dependency Injection Principles, Practices, and Patterns*. Manning, 2019. ISBN 9781638357100.

[29] Osinachi Deborah Segun-Falade, Olajide Soji Osundare, Wagobera Edgar Kedi, Patrick Azuka Okeleke, Tochukwu Ignatius Ijomah, and Oluwatosin Yetunde Abdul-Azeez. Developing crossplatform software applications to enhance compatibility across devices and systems. *Computer Science IT Research Journal*, 5(8):2040–2061, 08 2024. doi.org/10.51594/csitrj.v5i8.1491.

[30] Anna Skantz. Performance Evaluation of Kotlin Multiplatform Mobile and Native iOS Development in Swift. Master's thesis, KTH Royal Institute of Technology, July 2023.

[31] Frank Zammetti. *React Native: A Gentle Introduction*, pages 1–32. Apress, Berkeley, CA, 2018. ISBN 978-1-4842-3939-1. `doi.org/10.1007/978-1-4842-3939-1_1`.

[32] Liqiang Zhang. Teaching Cross-Platform Mobile Development and Cultivating Self-Directed Learners - A Six-Week Summer Online Course Experience. *J. Comput. Sci. Coll.*, 39(7):41–51, May 2024.

[33] Steffen Zimmermann. Migration From Native to Cross-Platform Development of Mobile Apps - Process and Prototypical Implementation. Master's thesis, Cologne University of Applied Sciences, 05 2021.

Index

Typeset December 18, 2024