Pedro
Azevedo

TÍTULO DA TESE (MÁXIMO 130 CARACTERES)

The Playground: mobile platform to connect amateur basketball players

DOCUMENTO PROVISÓRIO

Universidade de Aveiro
2025

Pedro Azevedo

TÍTULO DA TESE (MÁXIMO 130 CARACTERES)

The Playground: mobile platform to connect amateur basketball players

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor Ilídio Oliveira, Professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

agradecimentos / **acknowledgements**

Agradeço toda a ajuda a todos os meus colegas e companheiros.

palavras-chave texto livro, arquitetura, história, construção, materiais de construção, saber tradicional.

resumo Um resumo é um pequeno apanhado de um trabalho mais longo (como uma tese, dissertação ou trabalho de pesquisa). O resumo relata de forma concisa os objetivos e resultados da sua pesquisa, para que os leitores saibam exatamente o que se aborda no seu documento.
Embora a estrutura possa variar um pouco dependendo da sua área de estudo, o seu resumo deve descrever o propósito do seu trabalho, os métodos que você usou e as conclusões a que chegou.
Uma maneira comum de estruturar um resumo é usar a estrutura IMRaD. Isso significa:

- Introdução
- Métodos
- Resultados
- Discussão

Veja mais pormenores aqui:
`https://www.scribbr.com/dissertation/abstract/`

keywords textbook, architecture, history, construction, construction materials, traditional knowledge.

abstract

An abstract is a short summary of a longer work (such as a thesis, dissertation or research paper).
The abstract concisely reports the aims and outcomes of your research, so that readers know exactly what your paper is about.
Although the structure may vary slightly depending on your discipline, your abstract should describe the purpose of your work, the methods you've used, and the conclusions you've drawn.
One common way to structure your abstract is to use the IMRaD structure. This stands for:

- Introduction
- Methods
- Results
- Discussion

Check for more details here:
`https://www.scribbr.com/dissertation/abstract/`

Contents

List of Figures

List of Tables

Glossário

CHAPTER 1

Introduction

1.1 Concept

1.1.1 Basketball Context

Portugal is cleary not known for its basketball, the quality is far behind other european countries. In the International Basketball Federation (FIBA) world ranking**fiba**, Portugal is in the 47 place, and in the middle of the European table in 25th place. Comparing with our neighbours Spain that is in 7th in the world ranking. However, in the past years, the portuguese basketball achieve some marks.

First, Portugal had its first player in one of the best women league in the world, "Ticha" Penicheiro played during 15 seasons in the WNBA in th USA, winning a title with the Sacramento Monarchs, and some individual awards. And in 2019, entered the Women's Basketball Hall of Fame, that has the goal of honouring the most influential players in women basketball.**ticha**

After her, Neemias Queta was drafted to the NBA in 2021, being the first portuguese in the NBA, and in the 2023–24 season won his first title with the Boston Celtics. Besides not being the most valuable player in his team in the USA, he played an important role with the National team in second appearance in the EuroBasket tournament in 2025. In this campaign, the portuguese team made a surprise performance, passing through the group phase and then, confronting the German team. In this game the portuguese team was holding up until the last quarter, where the world champions and the winners of the EuroBasket 2025 took the victory. After the first victory in the EuroBasket, Neemias mentioned that it was a great moment fot the sport in Portugal and wants the sport to grow more in Portugal.**neemias** To help with this goal, a survey was made in order to understand how players and basketball enthusiasts use the public basketball courts, the frequency, and the problems.

1.1.2 Problem

After a some conversations with few basketball players and leaning heavily in a survey responses, basketball players and enthusiasts wish to go more often to public courts. Besides time constraints, this two scenarios and the following complains are in these players mind:

Key Scenarios.

- **Insufficient players**: people arrive to play but cannot form a team because not enough players are present;
- **Overcrowding**: courts are full, forcing players to wait or leave.

Common annoyances reported.

- Poor or unpredictable court conditions;
- Courts that are either empty or overcrowded;
- Social problems between players, including aggressive or unpleasant behaviour, individualistic style, and mismatched competitiveness (players who do not take the game seriously or whose skill and attitude creates a poor experience).

These issues point to a coordination and information problem. Basketball players lack a platform to discover, schedule and evaluate informal games, and to verify court availability and condition. As a result, players often waste time visiting courts with unsatisfiable condition, which decreases participation and lower quality of both social and sporting experience.

1.1.3 Proposed Solution

The proposed solution is to design and implement a digital platform that addresses the coordination and information gaps in amateur basketball in public courts, by focusing on two primary features:

Informal Games "Pickup games"[1] - Enable players to create, discover and join informal matches (pickup games), with features that support different play modes and social organization:

- Players can create, search for and join games listed on the platform;
- The games can be competitive or casual;
- Competitive games contribute to a leaderboard/raking table per court;
- Game formats include 1v1, 3v3, 4v4, 5v5;
- Users can create and mange persistent teams and challenge other teams;
- Users can set a team as "next"[2] to a game, to play against the winner.

Court Availability and condition awareness Provide users with real-time and information about courts so they can decide where and when to play:

- Persistent court catalogue with attributes - name, location, full/half court, number of courts, has water fountain, etc;

[1] "Pickup games" is the name used by basketball players to call a game without a formal organization (not in a league, no referee, no strick rules)

[2] "next" is used in the casual pickup games to tell the current teams playing, that wants to play against the winning team.

- Users can indicate intent to attend a court at a specific time, so others can have an idea of the availability;
- User can mark themselves "present" at a court;
- Optional automatic geolocation-based check-in to simplify presence signalling;
- Live occupancy indicators to show how many users intend to go and how many are currently present;
- Court condition reporting by the users, they can submit short status updates to report issues.
- Display short weather summary for the court location (optional)

1.2 State of the Technology

This platform has to reach the maximum basketball players for it to work properly, and should work when the players are in the court, or anywhere. In order to achieve this, the platform should be access from the mobile phones, and Statcounter[3] Global Stats reports the following about the Mobile Operating System Market Share in Portugal in September 2025: 65.65% is from Android, and 33.94% is iOS**statcount**. Despite the fact that more than half of people uses Android, the goal is still reach the maximum players possible, so the platform should be accessible from Android and iOS phones.

In order to develop a mobile application for both platforms, there are different options of approaches that can be selected for the implementation, which are the native, cross-platform, web, hybrid, modeling, cloud-based and merged approaches**Khachouch2020**.

1.2.1 Native Apps

Going Native, means that the app will be developed with to a specific OS, using its programming language. To build native Android applications, developers typically use Kotlin or Java. Both language are compiled to Java Bytecode that runs on the Android Runtime (ART). As for iOS, developers use Swift or Objective-C, mainly the first one, to build native iOS applications.

With native apps, there is no need for third-party frameworks, as the code is compiled directly into the platform's native language. This results in better performance and faster execution compare to other solutions, which will be discussed later. In addition to performance, native development offers security benefits, since developers can take advantage of the built-in security features of the operating system. When it comes to User Interface (UI) and User Experience (UX), each platform provides its own guidelines: Android follows Material Design, while iOS follows Human Interface guidelines. This consistency makes apps easier to navigate, and there is no issues in the app adaptation to other devices. Finally, native applications can make full use of device features such as the camera, microphone and GPS, which enhances functionality and integration.**Nagy2022**

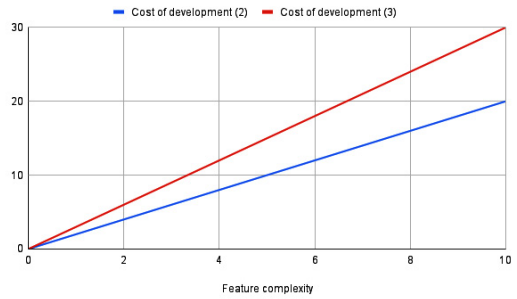
[3]Statcounter is a company that provides a tool for users to get insights of their website's visitors

When the goal is to deliver an application for both Android and iOS using native development, each feature must be implemented twice, once for each platform and in different programming languages. As Roby Nagy explains in *Simplifying Application Development with Kotlin Multiplatform Mobile***Nagy2022**, the development cost of this approach can be expressed as:

$$Costofdevelopment(n) = n * FC$$

where n represents the number of platforms and FC the *feature complexity*, defined as the sum of all sub-features that compose a feature. The figure 1.1 , taken from Roby Nagy' book**Nagy2022**, illustrates how the development cost grows for two platforms (blue line) and three platforms (red line) as the number of features increases, assuming no cost reduction from sharing code.

Figure 1.1: Cost of native development as a function of feature comlpexity **Nagy2022**



Beyond this calculation, additional factors complicate native development. Each platform has distinct characteristics, and a solution that is straightforward to implement on one platform may be unavailable or significantly harder to achieve on another. This divergence creates extra overhead in aligning features sets across platforms. So, Nagy introduces the additional component, *Synchronization Cost*, which increases in an exponential way as the number of features and its complexity grows. The updated formula is expressed as:

$$Costofdevelopment(n) = n * FC + SyncCost^{FC}$$

1.2.2 Web Applications

Any application that is access through *HTTP* requests over a network is called a web based application. These applications are implemented with HTML, CSS and JavaScript. Until Progressive Web App appeared, these applications were browser only, but now, they can be installed in the device, can have offline availability, push notifications and background sync, also able to run on desktop. Developers normally use Angular or React, and other frameworks are StencilJS or Svelte. However, PWAs cannot access key platform's features as the Native apps can, and also, has a possible higher energy consumption.**Huber2021**

1.2.3 Hybrid Applications

Another option, also leveraging web technologies such as HTML, CSS and JavaScript, is taking a hybrid approach. The app is developed with as web app and developers wrap the app via cross-platform wrappers, frameworks and tools, as a native app. With this approach, the app as a user experience just as a native app. Some hybrid app frameworks are Cordova, Phonegap, Titanium, Ionic, being the first the most common. Using third-party plugins with JavaScript, it is possible to access features form the device, such as the camera and GPS. In the Maher Gerges and Ahmed Elgalb article**Gerges2024**, where they compare different approaches, it is compared the hybrid apps directly with the native apps, and, in CPU and Memory occupancy, the hybrid cost more, being 106% higher and 73.0% higher in order. Also, not all features are available and the performance is worse than native apps.**Gerges2024**

1.2.4 Cross-Platform Apps

In order to avoid repeating the same code for different platforms, there are Cross-Platforms approaches. With this method, it is possible to create applications that run on multiple operating systems. In a first look, using cross-platform methods can be an effective solution to develop applications for a multiple platforms, when time, cost and basic interoperability is the primary concern. On the other hand, Cross-Platform applications tend to have the worst performance and user experience compared with Native apps, as they are behind the evolution of Android and iOS and cannot take full advantage of each platform's features**Nagy2022**. In different articles and papers, the categorization of the approach varies, especially, when it comes to the Cross-Platform approach, there are subcategories**ELKASSAS2017163**. The Web and Hybrid approach is normally inserted in this category, alongside with Interpreted approach and Cross-Compiled approach**ELKASSAS2017163**, which the most popular used frameworks are React Native and Flutter, respectively.

Interpreted Approach - React Native

Similar to the web and hybrid approach, uses web technologies such as CSS and JavaScript, however, the user interface is not HTML and not render in the browser.**Huber2021** The most popular used framework is React Native from 2015, developed by Meta, where developers build applications with React components, and the logic in JavaScript. This framework uses component-based architecture, following the steps of React for the web, promoting code reusability, modularity and easy debug.**10667693** The JavaScript code executes on a separate thread and communicates with the native layer through a bridge**Nagy2022**, allowing developers performance-critical parts of the app in native languages**10667693**. The bridge translates asynchronous, serializable data between the two environments, which enables native rendering of components. However, this approach introduces overhead, resulting in a process generally slower compared to fully native applications.**Nagy2022**

Cross-Compiled Approach - Flutter

In this approach, the app is developed in a common language, and then, it is compiled to the native code that can be executed on a mobile device, with Flutter and Xamarin popular

examples.**Huber2021** Flutter is a framework developed by Google, its architecture can be described in three layers, as noted by Roby Nagy in his book**Nagy2022**. The Framework layer, where developers write the application code and the UI components, declaratively using the Dart language. The Engine layer renders Flutter widgets to a canvas called Skia Canvas, which is then passed to the final layer, the native platform. The platform displays the canvas and sends back user events to the framework.**<empty citation>**Nagy2022 Compared to React Native, Flutter often achieves better performance by relying on Android's Native Development Kit and iOS's Low-Level Virtual Machine to compile code coming from the engine, Flutter applications deliver higher execution performance. Nevertheless, certain parts of a Flutter application may still need to be implemented in Java/Kotlin and Obj-C/Swift code. Communication between these native components and the Dart code is through a channel, reducing scalability**Nagy2022**.

Cloud-based Approach and Modeling Approach

A Cloud-based application, has its backend in the cloud, instead of running locally, using the cloud features such as flexibility, virtualization and security.**ELKASSAS2017163** Companies like Amazon Web Services, offer that service called Backend as a Service (BaaS).**Khachouch2020**

There are two subcategories in this approach, the Model-Based User Interface Development (MB-UID) and the Model-Driven Development (MDD). Developers use abstract model to describe tasks, data, users and the user interface, which will generate source code. It can transform to code from different platforms. Some tools include XMobile, for MB-UID and JavaScript Application Framework for MDD.

Cross-Platform vs Native

So, on one hand, using cross-platform frameworks can be an effective solution to develop applications for multiple platforms, when time, cost and basic interoperability is the primary concern. On the other hand, these frameworks tend to be behind the evolution of Android and iOS, and cannot take full advantage of each platform's features. Roby Nagy therefore refines the development-cost equation as:

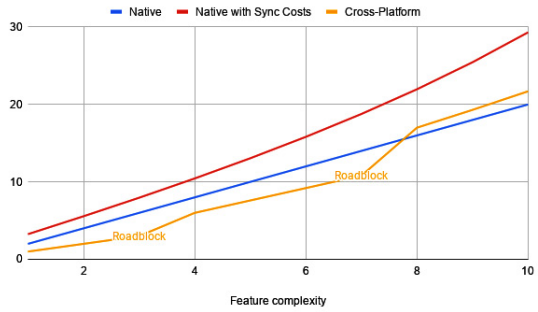
$$Costof Development(n) = FC * (1 + Costof goingNative)$$

In this equation, n is the number of platforms and FC is the feature complexity. As for the Cost of going Native, reflect the additional effort required when some parts must be implemented with native code. Its value depends on the level of interoperability that the cross-platform framework provides with native components, which is not optimal. When there is a necessity to implement native code, the knowledge gap between the framework and native languages adds to the expense, as well as the synchronization.**Nagy2022**

With this, it is possible to visualize the different cost of development, adding the Cross-Platform with roadblocks. The figure1.1, illustrates how the cost of cross-platform development

increases relative to both cost of Native and Native with Sync Costs. A roadblock occurs when a required feature cannot be implemented only with the frameworks and must instead be written in native code, producing a sudden rising in cost. If no native features are required, cross-platform development can remain less expensive with the previous discussed limitations. However, as more features demand native implementation, the cumulative cost eventually exceeds that of a fully native approach.**Nagy2022**

Figure 1.2: Cost of cross-platform development with potential roadblocks as a function complexity**Nagy2022**



1.2.5 Kotlin Multiplatform

Although cross-platforms the previous frameworks solve the problem of dealing with different platforms, it is still not optimal if the application demands platform specific features, and keeping up with Android and iOS updates. To go around these conditions, developers can choose Kotlin Multiplatform (KMP), also a cross-platform solution. As Jetbrains mentions in their website, Kotlin Multiplatform is an open-source technology developed by JetBrains, that allow developers to create applications for multiple platforms in an efficient way, by reusing code across them and still having the native approach benefits.[cite:@kmpJetbrain] Developers can share code without losing the qualities of native programming, have good user experience, good app performance and having full platform capabilities, alongside the benefits of the cross-platform, reducing development time, consistency in behavior in the different platforms. In this approach, the developer has 3 options, it can share and only write a critical feature in Kotlin, or implement the full logic and only develop the UI natively, and finally, with Compose Multiplatform, also share the UI.**kmp**

Jetpack Compose**jetpack** is a toolkit to develop the User Interface of Android apps. It is implemented in a declarative way, compatible with Kotlin APIs, enabling developers to intuitively build fast UIs. It also supports Google's Material Design, an "open-source design system for building beautiful, usable products"**materialD**. Since 2021, developers can build shared UIs for Android, iOS, desktop and web, feeling natural on every platform with Compose Multiplatform.**compose-multi**

Thompson Carter, refers in his book**kotlinInDepth** the same benefits of KMP, code reusability, faster development, reduce maintenance (as it is only need to fix bugs in one place), flexibility (share code and still access platform-specific features and APIs) and community support. However, he mentions some challenges as: - Maturity, as the KMP is recent and is

still evolving, some libraries and tools may not be fully supported; - The integration with Platform-Specific APIs can be challenging; - The transition from single-platform development can be a bit difficult, with the introduction of new paradigms; - And the integration with an existing project can involve significant refactoring.

1.2.6 Overview

Cloud-based / Model-driven Speeds backend development (BaaS); model-to-code generation can accelerate multi-platform delivery; centralized updates and services. Generated code quality and flexibility can vary; tooling maturity differs across vendors; vendor lock-in and integration complexity possible. For enterprise workflows, rapid prototypes, or when backend complexity and serverless integrations dominate.

Table 1.1: Comparison of mobile application development approaches (comprehensive).

Approach / Framework	Advantages	Disadvantages	Best when / notes
Native apps	Best performance; full access to platform APIs and sensors; highest UX fidelity; mature tooling and debugging.	Two codebases for multiple platforms with higher cost	When maximum performance, platform-specific UX or deep hardware access are required.
Web apps / PWAs	Single codebase; runs on desktops and mobile browsers; easy distribution via web.	Limited access to some native features; performance and energy use vary vs native;	When reach and ease of deployment matter; suitable for simple apps or when minimal native features are needed.
Hybrid (wrapped web)	Faster port from an existing web app to an installable app; single web codebase; plugin ecosystems to access device features.	Worse performance than native; higher CPU and memory usage; some native features may be unavailable; UX can feel non-native.	Quick conversion of existing web apps into installable apps; prototypes and low-budget MVPs.
React Native (interpreted)	High code reuse; React ecosystem; UI renders using native components; faster than pure WebView hybrids for UI responsiveness; good developer velocity.	Bridge overhead can reduce performance for heavy UI/CPU work; some features require native modules; platform parity depends on third-party library support.	Apps with standard UI and business logic where development speed and native look are both important.
Flutter (cross-compiled)	Compiled to native code (via Dart); high performance; consistent UI across platforms via Skia; rich widget set enabling polished UIs.	Larger binary sizes; occasionally needs platform channels to access native features; different paradigm (Dart and Flutter widgets) for developers.	When near-native speed is wanted and a single codebase with highly consistent UI across platforms.
Kotlin Multiplatform (KMP)	Share core business logic (networking, storage, domain rules) across platforms; retain native UIs or use Compose Multiplatform; near-native performance and good interoperability with native code.	Ecosystem still maturing; some libraries may be platform-limited; complex when integrating with existing native projects; tooling and third-party support are improving.	When substantial logic reuse is wanted while keeping native UI/UX and performance; especially attractive for Kotlin-centric teams.

Approach	Advantages / Features
Native apps	**Best performance**; **full access to platform APIs and sensors** (T
Web apps / PWAs	**Single codebase** (User interface code is reusable in multiple platforms
Hybrid (wrapped web)	**Faster port from an existing web app to an installable app**; **si**
React Native (interpreted)	**High code reuse** (Allows developers to write a single codebase for bo
Flutter (cross-compiled)	**Compiled to native code (via Dart)** (Compiles directly to ARM a

Table 1.2

APPENDIX A

Additional content