Seamless ticketing SDK

Fábio Rafael Gaudino
Caetano Pereira Pimenta
Instituto Superior Técnico
October 2021

ABSTRACT

The current ticket technologies used in public transportation revolve around smart cards. These smart cards are widely used but have some inherent faults associated such as easily being transferable between users, only being rechargeable in specific physical locations, and being required to be printed, making them not environmentally friendly. Hence, it is essential to develop an alternative. With smartphones being more prevalent then ever, it seems fitting to extend their functionalities to replace the usage of smart cards. In this dissertation, the Seamless SDK was developed alongside Card4B Systems, S.A., the Seamless SDK was designed to provide functionalities such as trip management and ticket validation to applications that integrate the SDK, using technologies like Bluetooth Low Energy for tracking the user inside the transport, which is crucial for the trip management, and QR code for acquiring the necessary information for the validation. The Seamless SDK was developed in a cross-platform framework to be integrated into applications in the two main mobile platforms, iOS, and Android.

Author Keywords

Public Transportation; Bluetooth Low Energy; QR code; Cross-platform development; Trip management; Ticket validation;

INTRODUCTION

Nowadays, technology is part of how society interacts with the world, and public transportation is no exception. In Portugal, most public transportation uses smart card technology to validate a trip. A few examples of this application are the Metro of Lisbon [1] and Vimeca [2], which operates many of the buses and the metro in Lisbon.

Smart cards were the right solution for ticketing for several years, but a more efficient solution is required in our day and age. Smartphones are more prevalent than ever, making them the ideal tool for ticketing since they can be used to validate and purchase a ticket.

Many transportation companies invested in developing mobile applications so that their users would be able to buy and validate their tickets through their application. By creating mobile applications with validation and ticketing mechanisms, the user can timely plan their trip, buy the ticket with their preferred payment option, and hold the ticket in their smartphone, ready to be validated.

This dissertation was developed alongside with the company Card4B Systems, S.A. [3], having the main objective of creating a Software Development Kit or SDK for short, that allows transports applications to use it for ticket validation and trip management in public transportation. This solution has the goal of providing a set of tools to be integrated into mobile transport applications. The SDK should facilitate the development of these applications by giving access to ready to use features needed in any ticketing system.

One of the key requirements for the development of this SDK, was that the SDK had to be usable and easily integrated in applications in both mobile platforms, Android and iOS. To this end, this dissertation will also have a substantial amount of research, analysis, and testing to cross-platforms capable of developing a shared SDK for both mobile platforms.

Another requirement was that the SDK uses technologies such as Bluetooth Low Energy, and QR code. This technologies are built-in on most mobile devices and are widely used in most transport applications.

With this technologies as foundations, the SDK will provide ticket validation and trip management in the following way:

The validation mechanism will consist of scanning a QR code present at the entrance of transport and gathering all the necessary information to be able to forward this information to a Card4B server to validate a ticket.

The other main functionality is a trip management module capable of tracking the user within the transport grid by detecting the Bluetooth Low Energy beacons inside the transport, as well as terminating a trip whenever requested.

STATE OF ART

CROSS-PLATFORM APPLICATION DEVELOPMENT

When developing a mobile application, there are different device platforms that you must build for, the more noticeable ones being iOS and Android. Traditionally, applications for each of these platforms must be constructed separately since each operating system uses a different code language not recognized by the others, this is called Native development. Cross-Platform Development provides a way to build an application using a common language, which can be run on both platforms. There are two possible approaches to Cross-Platform Development, which are Native and Hybrid cross-platform development [2]. We will focus on the Native

[1] https://www.metrolisboa.pt/
[2] https://www.vimeca.pt/
[3] https://www.card4b.pt/index.html

approach since Hybrid development is not suited for our solution. In this approach, the development of the application uses a framework with a common programming language and translates this development to the different device platforms with the use of Native API's. This results in an application that runs almost as fast as a Natively developed application and has access to the same elements but is compatible with both device platforms.

Frameworks

Cross-Platform App Development Frameworks are tools to develop applications. It consists of a software library that provides a fundamental structure to support development. There are many factors to consider when choosing a Cross-Platform App Development Framework. These influence the development and maintenance of the application. The factors that we will take into account are programming language, platforms supported, and longevity. We will also take into account the specific SDK properties of our project. Before looking closer into our factors, it is crucial to choose candidates frameworks to analyze and select the more appropriate one for our project. After close examination of many cross-platform frameworks, four primary candidates were selected for final deliberation: React Native [4], Flutter [5], Xamarin [6], Kotlin Multiplatform Mobile [7]. We will now look individually at our factors and analyze each framework by them.

Programming language

In terms of programming language, our main criteria will be time for accommodation to the language and support from the framework community. React Native is created by Facebook and uses JavaScript, which is the most used programming language in the world at the current time [15]. Personally, I have some experience with JavaScript, which will make the time for accommodating the language much shorter. React Native is also the most used framework worldwide. Being the most used framework also makes for the most significant community, which in return makes for more support from the community. Flutter is a reasonably new framework developed by Google that uses Dart programming language. Dart is a C style language, making it easier to understand someone with C style languages experience, as is the case. As Flutter, Dart is also rather new to the community, making both have smaller support than the pre-established frameworks. Xamarin is a framework developed by Microsoft which uses the C# language. The use of the C# makes for quick accommodation since its syntax is based on C++ and has influences from Java. Both of these were used in other projects by me. In terms of support, Xamarin has been active since 2013 and has created a respectful community in terms of numbers since then. Although not as big as the React Native community, both Xamarin and React Native allow for efficient use of the community help [9].

[4] https://reactnative.dev/
[5] https://flutter.dev/
[6] https://dotnet.microsoft.com/apps/xamarin
[7] https://kotlinlang.org/docs/kmm-overview.html

Kotlin Multiplatform Mobile, or KMM for short, is a recent and promising framework develop by JetBrains that is at the moment still in Alpha, it uses the Kotlin programing language that much like the C++ language, is greatly influenced by Java. Kotlin is also one of the most used languages for native android development and although KMM is very recent, the Kotlin language was release in 2011, being a fairly well established language. Due for how new KMM is, the community around Kotlin Multiplatform Mobile is also very small.

Platforms Supported

In terms of platforms supported, the frameworks support at current time the versions showed in following table 1 [8] [10] [3] [7].

Framework	iOS version supported	Android version supported
React Native	iOS 10.0 or newer	Android 4.1 (API 16) or newer
Xamarin	iOS 9.0 or newer	Android 5.0 (API 21) or newer
Flutter	iOS 8.0 or newer	Android 4.1 (API 16) or newer
KMM	iOS 7.0 or newer	All versions

Table 1. Frameworks comparison

In both iOS [12] and Android [13], the four frameworks can be used in more than 99 percent of devices, which means all four give almost perfect compatibility with the mobile platforms.

Although not a differentiator factor, the version supported for which platform is a critical requirement in any cross-platform framework. The inability to support almost all the versions used leads to many users not being able to use the SDK developed in this thesis, which in turn makes the selection of a framework without this characteristic unsatisfactory.

Longevity

When analyzing the possible longevity of a framework, the two main ways to assess it are the number of users and time since launch. These two will give us some insight on the chance of the framework being discontinued. React Native leads the way in the number of users in 2020 and 2019 [11], maintaining a substantial share of cross-platform developers and, as a result, the number of applications developed. It was released in March 2015, having time to mature as a framework, making it the least likely to be discontinued in coming years. Flutter has seen a rise in popularity in the last two years, especially in 2020, reaching a similar percentage to React Native. Flutter was launched in May 2017, making it a relatively recent cross-platform framework. Which could mean that the software did not have enough time to fix some vital compatibility issues [4].

Xamarin is the opposite case of Flutter. It was seen a downfall in use and popularity in the last two years. Xamarin can develop cross-platform applications since May 2014, making it the oldest framework of the four [1].

Kotlin Multiplatform Mobile was a similar longevity to Flutter, being released in November 2017. But contrary to Flutter, it has a much lower usage percentage [14], making it the least used framework of the four.

Selecting the Framework

The selection of the framework took into consideration the above characteristics of the four analyzed frameworks. The selected framework in the early phase of the project was React Native due for the fact of excelling in all three of the criteria, longevity, compatibility, and programming language. React Native seemed the best option by far. This framework was tested thoroughly to see if it had the capabilities to create an SDK to be used in both Android and iOS systems. After exhaustive tests and research, it was concluded that the separation of the created SDK and the target application that uses the SDK is not possible. All the outputs created with React Native need to be executed with a React client. This makes the integration with other native applications impossible. The only possible way would be creating both the SDK and target application in React Native, defeating the purpose of creating an SDK that does not have integration capabilities with other native applications.

This lead to the investigation of other alternatives, Flutter and Xamarim were considered, but their capabilities and problems were similar to that of React Native. Since this frameworks could not create the separation needed of an SDK, the focus of the research was switched to KMM. When researching the capabilities of KMM, it was noticed that this was the only framework that actually generates native binaries for the target mobile platform, this meant that in theory KMM could generate a native SDK to which one of the mobile platform while maintaining only one codebase and providing the necessary separation of an SDK. It was at this point that Kotlin Multiplatform Mobile was chosen.

Kotlin Multiplatform Mobile

As said before, Kotlin Multiplatform Mobile is a recent and promising framework for cross-platform development released in November 2017. KMM is still in alpha, meaning that its development is growing rapidly but is not stabilized yet. Although very new, KMM is based on the programming language Kotlin, which was over a decade of existence and is widely used as a more concise and secure language alternative to Java in application development. KMM works by using the multiplatform abilities of Kotlin and the features designed for mobile development in which specif platform (iOS and Android).

Kotlin Multiplatform Mobile is designed to help develop a single codebase for persistence and business layers in Kotlin, leaving the UI and some of the presentation layer for native specific code. This makes KMM a perfect fit for our project since our SDK functionalities are mainly focused on the persistence and business layer.

Architecture of Kotlin Multiplatform Mobile

As said before, KMM is a tool that allows sharing a single codebase of business logic and back-end operations with iOS and Android applications, maintaining the freedom of writing platform-specific code whenever is needed. As we can see in Figure 1, the shared code will be used by both iOS and Android applications while maintaining code specific in the same layer of KMM but separated from shared code. This architecture also maintains a layer between the KMM code and the applications which is why KMM can be integrated where other frameworks such as React Native could not.

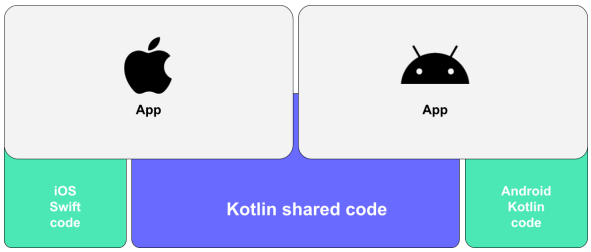

Figure 1. KMM Architecture

To further clarify the architecture of Kotlin Multiplatform Mobile, we will present the development environment and the folder system used in the creation of our project, as showed in Figure 2.

The Shared folder is subdivided into three folders commomMain, androidMain and iosMain. In the commonMain folder, we can find abstract Kotlin code that does not use specif dependencies from iOS nor android. This folder is where we can find all of the business logic-related code. This folder is where most of our work will be done. The androidMain and iosMain folders fulfill the same purpose for their corresponding platforms. Their purpose is to write platform-dependent Kotlin code, such as manipulation of platform-specific objects. This code can then be called from the commonMain folder depending on which platform the shared code is running.

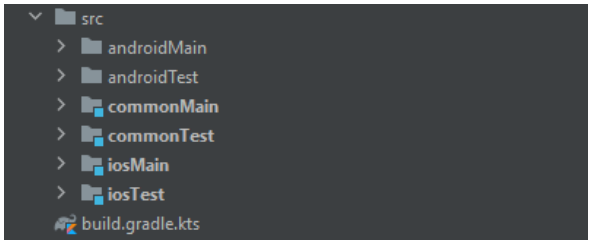

Figure 2. KMM folder system

ARCHITECTURE

SEAMLESS SDK

Seamless SDK is a software development kit that offers a collection of functionalities that allows the development of applications for the seamless usage of public transportation. Our Seamless SDK assumes some pre-established physical components in public transportation, which are the existence of BLE beacons inside the transport capable of transmitting information to the user device and an QR code in each entry point of the transportation infrastructure.

SEAMLESS SDK DESIGN

We will now go into an overview of the Seamless SDK usage and surrounding components. We start with the interaction between the user and the application (a). As shown in the Figure 3, the SDK is a component of the application, which means that all interactions between the user and SDK will be mediated by the application in the form of requests.

When the application requests some SDK functionality, the Seamless SDK will communicate with the server/API (b) to either get information, report some event, or store data. If the user requests the start of a trip, the validation process will be triggered d). The validation mechanism consists of using the user smartphone camera to read a QR-code present at the transport. The information retrieved will be used by the SDK for validation.

During the trip, the Seamless SDK will actively listen to the surrounding BLE beacons signal c) to know in which station the user is present.

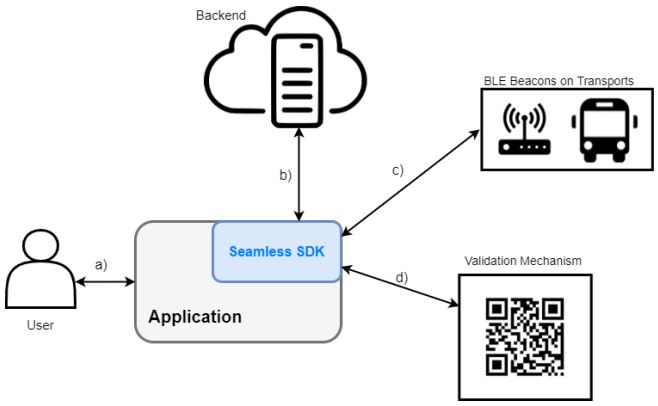

Figure 3. Seamless SDK design.

ARCHITECTURE LAYERS

The Seamless SDK architecture will be described based on Figure 4 and three layers of abstraction to better understand the architecture, them being the View, Presenter and Model layers. These layers will contain the various modules and components necessary for the proper behaviour of which level. These layers are part of a standard architecture pattern called MVP. The MVP architecture was chosen because of its modularity and maintainability. Both of these are vital factors taking into consideration the nature of our project.

Modularity is fundamental for the replacement and integration of modules quickly, which is very common in an SDK to be implemented in different applications. Maintainability is a must-have in a project that implements Kotlin Multiplatform since platform specif conditions can make the codebase confusing. By implementing an architecture pattern that makes the codebase more clean and maintainable counterbalances this aspect of KMM.

View Layer

The View layer is responsible for interacting with the user, as described in Figure 4, making it a fundamental part of our

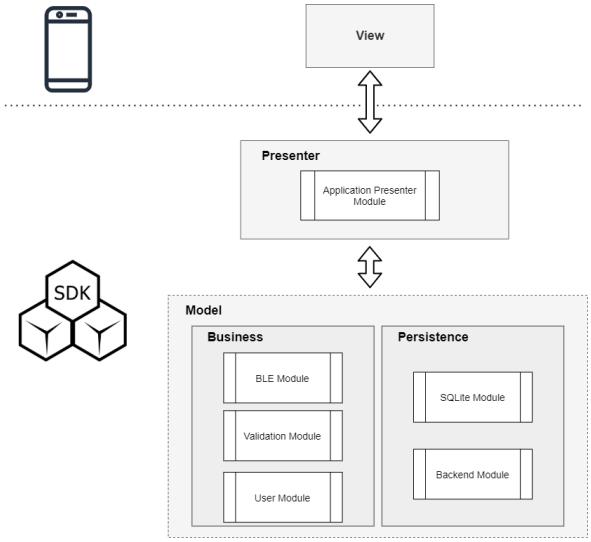

Figure 4. Seamless SDK Architecture

Architecture. This layer is implemented via an interface in the application where our Seamless SDK runs. It is on this layer where the UI of the application is and where all of the user's requests originate. This request is then forwarded to the presenter.

Presenter Layer

The Presenter layer is responsible for the communication between the View and the Model Layer, acting as the middleman. In this layer, we can found our presentation module. This module will act as an intermediary between the functionalities of the SDK and the application's requests. The application will initialize the module in the View Layer and gain control of the View's data. Whenever a request is made to the SDK, it is made through this module that will forward the request to the Model layer. After the request is processed, the presentation module requests some UI alterations or responds to the application if no changes are needed in the UI.

Model Business Layer

The second layer is the Model Business layer. This layer is where most of the data manipulation, business rules and technology specif usage are. In the Model Business layer, we can find three modules, the BLE module, the validation module, and the user module.

Starting with the BLE module, this module monitors the BLE beacons around the user device. This monitoring has quite a few usages and tasks associated with it. The first one is making sure that a BLE device found makes part of the grid of the public transportation devices that the application is inserted. In that endeavour, the device's ID is compared to the one the SDK possesses of that particular station, asserting if the device is valid. Monitoring the BLE devices also has the purpose of maintaining a list of stations that the user passed on so that the Seamless SDK can return a list of stations passed during a user trip.

The second module is the validation module. This module starts the camera of the user smartphone and scans the QR code present at the transportation entrance for the necessary information for the validation of the trip. This module also makes some confirmations regarding the location of the QR code with the help of the BLE module.

In the BLE and validation modules, there is a division of code related to the usage of the KMM. This division is necessary due to the fact that the technologies used on these modules are platform-specif technologies, which means that each has to be coded in their specif platform folder (iosMain or androidMain). Although some code is written in their platform-specif folders, the modules are created with a Kotlin shared base that manages all of the data manipulation and requests from other parts of the Seamless SDK and the module itself.

The third and final module is the user module. This module works closely with the backend module, which we introduce in the next layer. The primary responsibility of the user module is manipulating the user data received from the back end module to be handed to the presenter layer.

Model Persistence Layer

The last layer of SDK is the Model persistence layer. On this layer is where we can found the SQLite module and the Backend module.

The SQLite module is where we perform the operations of storing and retrieving data from our local database. This module uses SQLite local database, which is a database that is located in the user smartphone.

The other module present in the persistence layer is the Backend module, where all the calls to the server are made. The primary purpose of this module is to consume a REST API of a list of users and their various information's. This module works as a proof of concept in order to see if it is possible to create a shared module of a working HTTP client for both mobile platforms.

IMPLEMENTATION

VIEW LAYER

Although there is no module implemented in the View Layer, the Seamless SDK contains some interfaces to be implemented in this layer for the correct interaction between the View and Presenter Layer. These Interfaces have a set of functions for the manipulation of the UI of the application. After implementing these functions, they will be called from the Presenter Layer when a modification is required.

PRESENTER LAYER

The presenter module is present in every other module that interacts with the UI of the application. It is a class that is implemented to which module to interact with the application and requests the modification of the UI for which individual module. For example, the BeaconListPresenter is a class that is instantiated in the View layer and then given the view class of the current activity of the application, making the BeaconListPresenter able to change the current View through the use of the functions implemented in the view layer described above. So, in this case, the BeaconListPresenter can forward a request from the user to the BLE module, receive a response from the BLE module, that would be a list of the surrounding BLE beacons, the presenter can then present it by calling the function "showBLEDevices" implemented in the View Layer.

MODEL BUSINESS LAYER

BLE Module

The BLE module has the general purpose of monitoring the BLE devices around the user device and confirms if they are part of the grid of BLE devices of the transport. To this end, it was implemented the android beacon library Altbeacon. With this library, we can assess the beacons that come close to the user device by maintaining a call to a function with a one-second frequency to estimate new beacons in the range of the user device. This library also allows setting which type of beacon layout we want to identify in the search. In our case, the Eddystone UID layout was chosen. A BLE device can take different roles for different purposes. In our system, the roles will be Broadcaster for the BLE devices found on the transport infrastructure and Observer for the user's smartphone. This role pair implements unidirectional, connection-less communications, meaning that all communications are one-way, from the Broadcaster to the Observer. This also means that a connection is not established between the devices. Our Broadcaster will periodically send advertising packets with data, following the Eddystone UID format.

Eddystone is a BLE profile that contains several frame types, including the UID format. This format can be divided into two parts, as described in Figure 5, the first part is the Prefix Data. The prefix is present in every format to identify the frame as Eddystone format and has a size of 11 Bytes. The UID frame data has a total of 20 Bytes and is divided into three subsets of data. The first 4 Bytes is where we can find the Frame type, in this case, the UID, and the Ranging Data that gives us the distance in meters between the Broadcaster and the Observer. The other two subsets together are, as mention in section , the universally unique identifier or UUID of the BLE device. As mention before, the UUID is a 128-bit number or 16 Bytes that uniquely identify a beacon. The UUID is composed of two subsets of data: the Namespace with 10 Bytes and the Instance with 6 Bytes. These two subsets will allow obtaining various pieces of information about the transport.

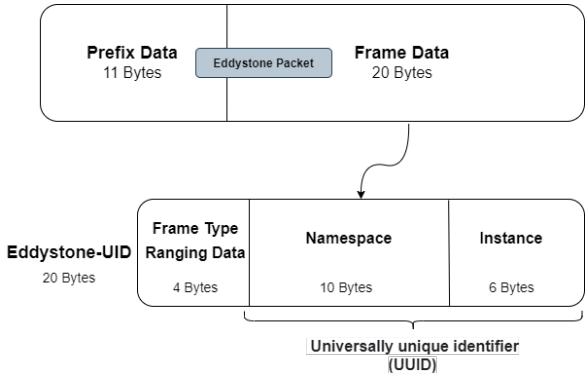

Figure 5. Eddystone-UID Frame

Since both the Namespace and Instance are customizable, we can place information about the location of which individual beacon in these fields. The information deemed necessary for the proper identification of each beacon is described in Figure 6. Depending on the type of transport, more data may be required. With that in mind, the 16 Bytes were not fully used.

For now, the fields to be advertised in the UUID will be the beacon id, the stop in which the beacon is, and the line id of the stop.

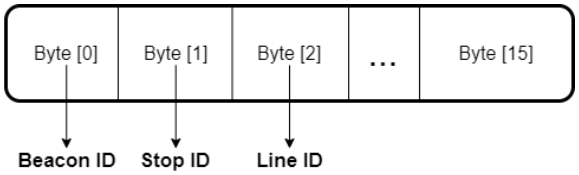

Figure 6. UUID Fields

By scanning which beacon the user device passes, the BLE module can collect information from these fields and create a list of stops that the user passed by. This list will contain the beacons found sequentially, and which beacon will have the time it was found, creating a detailed route that the user used during his trip.

This module is responsible for three of our functional requirements, them being FR1 manage the state of the trip being performed, FR2 provides information about the transportation, such as lines our routes, and FR3 yield information about current BLE beacons in range.

Validation Module

The validation module is responsible for starting a trip and gathering the necessary information for the validation of a trip . This module uses the library codescanner. With this library, the module is able to read the content of a QR-code. The module starts when a user requests the start of a trip, this request is sent to the validation module, and the module will use the camera of the user device to initialize the scanning of a QR-code. After the QR-code is scanned, the camera preview will stop. The information present in the QR-code placed in the transport will be a unique identifier for the QR-code and the station identifier in which the QR-code is present. This information will then be confirmed against the surrounding BLE beacons with the help of the BLE module. If the nearest beacon has the same station Id as the QR-code and the QR-code identifier is part of the corresponding station, the trip will be validated and proceed to the next step. The functional requirement that the validation module is responsible for is FR4 validation of a trip.

User Module

The user module was implemented to facilitate the processing of the users data that originates from the backend module. This processing is responsible for mapping the received data to a usable data class and introduce the necessary attributes to the presenter layer of this module. Since this module is composed solely of business rules and data classes for the manipulation of data, it is exclusively written in the commonMain folder, making it a truly shared module.

MODEL PERSISTENCE LAYER

SQLite Module

This module is responsible for creating and managing our local SQLite database. Its main functionalities are fetching and storing all data that requires a certain level of persistence in our local database. Our implementation relies on the SQLDelight library, which is the standard in our Kotlin Multiplatform Mobile environment. It has a simple implementation, which requires few lines of code in the platform-specific folders iosMain and androidMain. The SQLDelight library generates a database class associated with our Schema, which is a collection of SQL objects such as tables and queries. The database class can be used to create our local database and run the type-safe functions generated by SQLDelight. These functions are generated at runtime from the queries defined in our Schema and can be used directly in Kotlin.

This local database main functionality is allowing to maintain a persistent map of the transport grid, to be used in the validation and BLE modules.

Backend Module

As specified in our architecture, the Backend module is responsible for communicating with an API and consuming a list of users. This module uses ktor, which is a framework for networking. With it, we can build asynchronous clients and communicate with servers. This framework was selected, much like SQLDelight, because it is the standard in the Kotlin Multiplatform Mobile environment and is very well documented.

The implementation of a ktor HTTP client requires no code in the platform specif folders, meaning that the module is fully implemented in the commonMain folder. To communicate with a server, we create our HttpClient class and configure it with the JSON features necessary for our response body's proper serialization and parsing. After our HttpClient is created and configured, we can make our GET request to the server. In order to make this request, we first need to create a coroutine, which is conceptually similar to a thread meaning that it runs a block of code concurrently with the rest of the code. By executing a coroutine, we can make asynchronous calls, as is the case with our GET request. The result of our GET request is a list of fifty users, which user is composed of the following attributes: id, first name, last name, username, email, profile picture URL and gender. These users are then forwarded to the User module upon request. The API used is a public API that generates a random set of users, and the size of this set can be changed according to the URL provided in the GET request.

DEPLOYMENT

Our deployment follows three significant steps, transforming our KMM project to a library, compile the project into his native binaries and integrating the output to a native demo application.

The second and third steps differ depending on which type of platform they are being targeted to. So with that in mind, these steps will have two subcategories for Android and iOS, respectively.

The first step was to transform our KMM project from a running module to a library. In order to do this, some alterations were made to the build.gradle file, this file is the build configuration script of our project. The alterations consisted of changing the task of compiling and building our project from an application to a library and adding some required dependencies for the libraries used in our project.

Secondly, it was necessary to change the targets to which our SDK was compiled to. This is one of the KMM functionalities called Kotlin/Native, which compiles Kotlin code to native binaries.

Android:

The second step is relatively straightforward in Android, in the build.gradle file, we add in the library extension configurations the API level of our system and the target API level we are compiling. After doing this, we can build our project, creating an AAR, which is a bundle usable in Android applications. This output is our packaged SDK.

iOS:

In iOS, the second step involves a bit more effort. First and foremost, we require an iOS system to create our binaries, so the project must be transferred to a macOS, for example. After transferring the project and adjusting the gradle JVM to the project, in our case, we used the corretto 11.

After setting up these configurations, we need to specify to which platform we are compiling our project too. We can select iOSX64, which is used for iOS simulators, and iOSArm64 used in physical iOS devices. In our case, we will compile for both, and to each, we give a name to identify that particular output. Finally, we run the build option, and our binaries are created in the form of a framework. A framework is a hierarchical directory that encapsulates shared resources, as is the case of our SDK. The framework folder will have the name structured as follows "nameGiven".framework.

In our third and final step, we need to integrate our Seamless SDK in our native application demo.

Android:

To add our AAR file to the demo application, we need to copy our AAR file to the libraries folders of our demo application and add the SDK as one of our dependencies. To use the resources of our SDK in the application, all we need is to sync our project and import the resource we want from the Seamless SDK to the code in the application.

iOS:

The third step is quite simple in iOS as well. First, we need to add our framework folder to the iOS project folder. After adding the framework, we need to go to the xcodeproj file, which is a file that contains all the information regarding the location of the source code as well as the usage of libraries and frameworks. In this file, in the frameworks and libraries section, we need to add the folder of our framework. Doing this will make our Seamless SDK usable by importing it into the code.

With these steps, a successful deployment can be made of the Seamless SDK to both Android and iOS applications.

EVALUATION

IOS LIMITATIONS

As mentioned above there are a few limitations regarding the iOS mobile platform that arose during the development of the BLE and validation modules. This limitations are caused by the Kotlin Multiplatform mobile framework, as described in section 3.3, KMM is a very recent framework making it susceptible to some limitations to a more specific project like our Seamless SDK. The limitations originate from our iosMain folder where we write native Kotlin code that is able to run in the iOS environment, this is a really powerful tool and works perfectly with the main libraries of iOS that have been migrated and adapted within the KMM framework. The problems start when we require a library outside the scope of the libraries provided by KMM as is the case with both of our modules. Many hours of research were made in order to find possible solutions for the external library problem, two possible solutions were found but due for their complexity and for the time constraint of the dissertation, neither were implemented. Nevertheless, the two solutions will be described since the future of this project is directly correlated with overcoming these limitations and a formal description of our founded results will save time in future developments.

To use a external library in the Seamless SDK there are two possible options:

- Writing native Kotlin code to call functions in Objective C libraries.
- Creating a Kotlin function with a callback from Objective C / Swift that originates from the target application.

Starting with the first one, we must first import the Objective C library to our project, this in itself has a level of complexity but Kotlin provides some documentation on the subject [5], it is also important to note that only Objective C libraries can be imported, meaning that it is not possible to use pure Swift libraries in this solution. The next step is creating bindings for the external library, this bindings work like a bridge between the native Kotlin code and the Objective C library. Although the bindings are written in native Kotlin they are representations of Objective C classes, meaning that writing these is almost like learning a new programming language making it a extremely complex and a time demanding process, there is also some documentation regarding this subject [6]. After the last step we can use the functionalities of the external library.

The second option consists of creating a separated native module written in Swift or Objective C that uses the external libraries and importing this module to the target application. The SDK will then communicate with this module via callbacks using the target application as a middle ground between the two. Although this solution is much simpler, it also has

drawbacks, the mains ones being that the SDK is not encapsulated in just a single fully operational output and that the target applications needs to handle some integration between the native module and the SDK.

DEMO APPLICATIONS

In order to test our Seamless SDK functionalities, we first developed the Android application for two main reasons, them being that android development is within my knowledge scope, and the integration of the Seamless SDK with the android environment is more straight-forwarded. Initially, it was also presumed that some limitations could appear in the iOS environment, like the one described in section 15, which was also a factor when deciding which mobile testing environment to develop first.

Android Application

The Android application first screen displays two options, the first one with a button with the text "Fetch Test Users" and the second one, a button with the text "QR Code Validation" The first button will go to another screen, where a list of users is showed, which user is composed of is first and last name and their email. In this screen, the test application calls the user and backend modules. As described in section 12.2, the backend module retrieves a list of users of a public server, and the user module manipulates and delivers the users to the View layer in a presentable fashion. On the other hand, if the second button of the first screen is pressed, the application will start the smartphone camera to validate a QR code, as shown in Figure 7. This screen is where the validation module is activated and tries to validate the QR code.

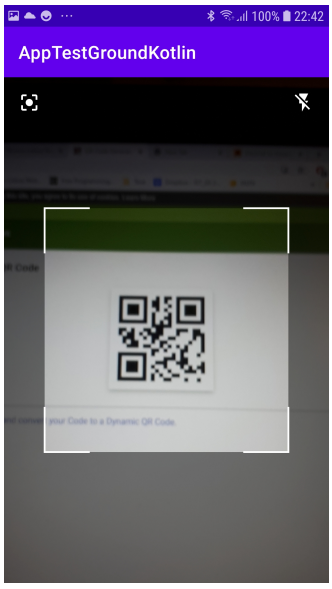

Figure 7. Demo android application QR code.

If the QR code is accepted, the application will go to the next screen, also showed in Figure 8. In this screen, the application initializes the BLE module and requests a list of nearby BLE beacons that are then shown on the screen. Which beacon presented on-screen displays it BeaconId, StopId and Station Name. At the bottom of this screen, we can also see the trip being performed. This trip has two buttons, the "?" and "X" buttons. The "?" button provides extra information about the trip. The "X" button will request the BLE module to stop the trip on the last station added, specified in the extra information about the trip.

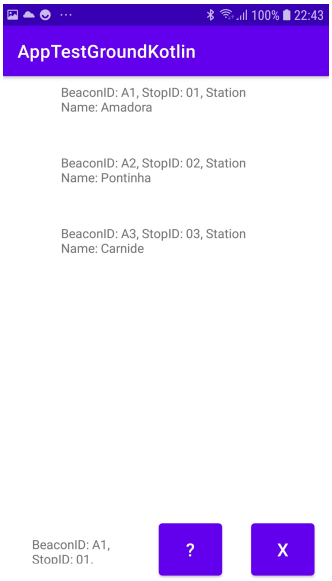

Figure 8. Demo android application BLE list and Trip.

iOS Application

The iOS demo application only has one screen that is depicted in Figure 9. The screen shows a list of the fifty generated users, provided by the Seamless SDK backend and users modules.

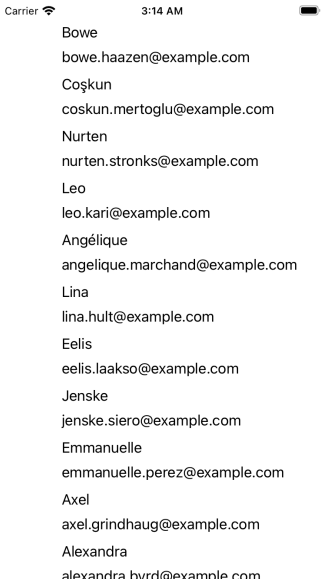

Figure 9. Demo iOS application Users List.

Also in this screen, is where the application initializes and retrieves data from the local database through the SQLite module.

The two demo applications developed in this dissertation provided some results and insight about the Seamless SDK and the cross-platform framework we used. The main result extracted is that it is possible to develop a shared codebase SDK in KMM and distributed it to different applications, this result is one of the primary goals of this dissertation.

SHARED CODE ANALYSIS

The use of Kotlin Multiplatform Mobile in the development of the Seamless SDK brought some questions during the development of how compatible the shared files would be with the native applications and how much specific code would have to be written in the androidMain and the iosMain folders. With this in mind, it was performed a count of how many lines of code were produced in which folder, in order to analyse the usefulness of the Kotlin Multiplatform Mobile framework in the development of the Seamless SDK and extrapolate this results for similar projects.

Although lines of code are not the most reliable metric if we require a correlation to time, since defining a proper conversion between the two is practically impossible, comparing the number of lines of code between two possible projects that achieve the same result as some merit in identifying the effectiveness and usefulness of a toll in the scope of cross platform development.

The percentages extracted from the Seamless SDK are displayed in Figure 10, the number of lines for which category was 917 for the shared folder (commomMain), 204 for the android folder (androidMain), and 114 for the iOS folder (iosMain).

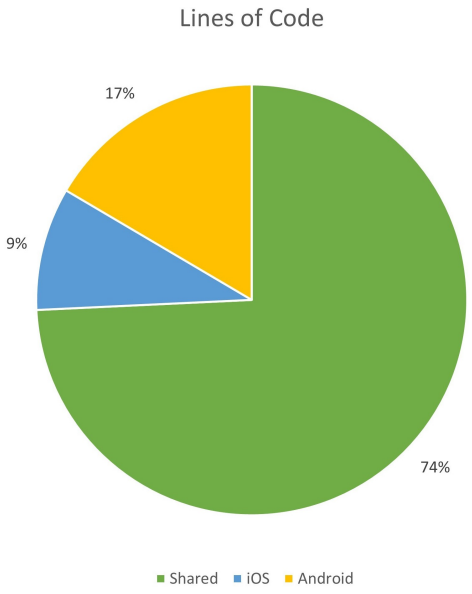

Figure 10. Pie Chart of Lines of Code.

It is important to note that due to limitations of the iOS referred in section 15, the line count of the iOS folder may not fully represent the complete Seamless SDK number of lines. But taking as an example the android folder percentage, we can presume that the iOS folder would follow the same trend of percentages since the implemented components are mostly comparable.

	Lines of Code	Improvement compared to Seamless SDK	Improvement Percentage
Seamless SDK	1235	-	-
Android SDK	1121	114	+10,7%
iOS SDK	1031	204	+19,8%
Android + iOS SDK	2152	-917	-42,6%

Table 2. Comparison example of developing a SDK in KMM against Native

With the previous statement we can also assume that the shared percentage would range between 65 to 70 percent. To better comprehend this result, we will make a simple calculation to estimate how many lines of code would have to be written to create two Seamless SDK's for their respective mobile platforms and languages. For the Android SDK, the line count would be the sum of the lines in the shared and android folder to a total of 1121 lines. As for the iOS SDK the logic applied would be the same, making the sum of the iOS folder and the shared folder, for the total of 1031 lines. By adding both, we get a total of 2152 compared to the original 1235 lines, effectively writing more 917 lines of code or 42,6 percent more lines in total, this example is represented in table 2. This example demonstrates how useful the 74 percent of shared code really can be in a project.

But not every project follows the exact proportions of business rules to technology implementation as our Seamless SDK. By analysing our Seamless SDK, it became apparent that the code present in the iOS and android folders are implementations related to the peripherals, such as the BLE and QR code, almost exclusively. A project composed predominantly of business rules, such as the user module, would have a much higher percentage of shared code. A project consisting mainly of peripherals implementations would have a much lower shared code percentage.

This percentage of shared code also dramatically influences the project's maintainability since the higher the percentage of shared code, the lower the likelihood is of a developer maintaining two different implementations of a feature in the iOS and Android folders.

CONCLUSION

The approach used nowadays for ticket validation in public transports is largely based in smart card, most of the smart cards shortcomings can be overcome by adopting new validation and ticketing mechanisms and applying them to smartphones. With this in mind, it was idealized the development of an SDK that would be able to provide trip management and ticket validation to Android and iOS applications.

This dissertation was developed alongside Card4B Systems, S.A., and began with the research of the technologies that were used in the development of our solution. The BLE technology is a cornerstone technology for our SDK that provided a way to track the smartphone inside the transport grid, to this end we research the various communication mechanisms so we could use this technology. One of fundamental goals of the SDK is to be usable in Android and iOS applications and as suggested by Card4B we started the research of cross-platform development, the study of this topic was exhaustive since all of our solution depended of these findings, it were analyzed four different cross-platform frameworks, and after several tests we selected the Kotlin Multiplatform Mobile framework.

The Seamless SDK is a software development kit that provides seamless trip management and ticket validation functionalities to transport applications that integrate it. The Seamless SDK was designed taking into consideration the various possible transport scenarios, making it flexible enough to be adapted into different contexts. The SDK was architected into layers comprised of different modules, which one with different responsibilities, the modularization of the architecture has the goal of making the SDK easy to maintain and adapt. After implementing the Seamless SDK, we reach a solution capable of gathering all the information necessary for a ticket validation through the scan of a QR code, track a user trip by scanning the BLE beacons around the smartphone, manage a trip being performed, communicate with an external server and storing information in the local database. The next step was deploying and integrating the Seamless SDK into different applications, during this step we elaborated a procedure for future use.

To evaluate the Seamless SDK two demo applications were developed, one in Android and another in iOS, these applications prove that is possible to integrate the Seamless SDK into both mobile platforms as well as showing that the SDK functionalities have the proper behaviour. This being said, we also described some of limitations of the Seamless SDK in the iOS platform that originate from the KMM framework. It was also evaluated the usefulness of KMM in the development of the Seamless SDK compared to native development.

To concluded, the results of the development of Seamless SDK are positive, we successfully created a solution capable of being integrated in both mobile platforms while only having one codebase and providing the basic functionalities for trip management and ticket validation for an transport application. Although not complete due for the limitations referenced previously, it was suggested some possible solutions for this shortcomings that will make the Seamless SDK much closer to implementation in actively used transport applications.

REFERENCES

[1] Altexsoft. 2020. The Good and The Bad of Xamarin Mobile Development. https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native/. (2020). Online; accessed 26 December 2020.

[2] CLEARTECH. 2020. What is Cross Platform Development? https://www.cleart.com/what-is-cross-platform-development.html. (2020). Online; accessed 21 December 2020.

[3] Flutter. 2020. What devices and OS versions does Flutter run on? https://flutter.dev/docs/resources/faq. (2020). Online; accessed 22 December 2020.

[4] Github. 2020. Flutter Issues. https://github.com/flutter/flutter/issues. (2020). Online; accessed 26 December 2020.

[5] Kotlin. 2021a. Add dependencies to KMM modules. https://kotlinlang.org/docs/kmm-add-dependencies.html#ios-dependencies. (2021). Online; accessed 27 September 2021.

[6] Kotlin. 2021b. Interoperability with C, Bindings. https://kotlinlang.org/docs/native-c-interop.html#bindings. (2021). Online; accessed 27 September 2021.

[7] Kotlin. 2021c. Supported platforms. https://kotlinlang.org/docs/kmm-overview.html#supported-platforms. (2021). Online; accessed 14 September 2021.

[8] Microsoft. 2020. Xamarin.Forms supported platforms. https://docs.microsoft.com/en-us/xamarin/get-started/supported-platforms?tabs=windows. (2020). Online; accessed 22 December 2020.

[9] Radixweb. 2020. Choosing the Best Cross Platform Mobile App Development Framework. https://radixweb.com/blog/cross-platform-app-development-frameworks. (2020). Online; accessed 23 December 2020.

[10] React Native. 2020. Requirements. https://github.com/facebook/react-native#-requirements. (2020). Online; accessed 22 December 2020.

[11] Statista. 2020a. Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. (2020). Online; accessed 26 December 2020.

[12] Statista. 2020b. Mobile Android operating system market share by version worldwide from January 2018 to January 2020. https://www.statista.com/statistics/921152/mobile-android-version-share-worldwide/. (2020). Online; accessed 23 December 2020.

[13] Statista. 2020c. Mobile Apple iOS operating system market share by version worldwide from January 2018 to April 2020. https://www.statista.com/statistics/1118925/mobile-apple-ios-version-share-worldwide/. (2020). Online; accessed 23 December 2020.

[14] Statista. 2021a. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. (2021). Online; accessed 15 September 2021.

[15] Statista. 2021b. Most used programming languages among developers worldwide, as of 2021. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. (2021). Online; accessed 12 September 2021.