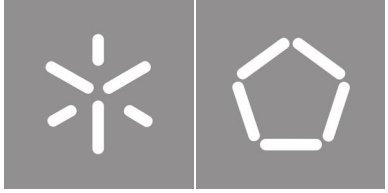
Universidade do Minho
Escola de Engenharia

Carlos Miguel Rebelo Solans

iOS Development: Increasing Home Banking Reliability with Integration of Strong Authentication Mechanism

November, 2021



Universidade do Minho
Escola de Engenharia

Carlos Miguel Rebelo Solans

iOS Development: Increasing Home Banking Reliability with Integration of Strong Authentication Mechanism

Master Thesis
Master in Informatics Engineering

Work developed under the supervision of:
José Carlos Leite Ramalho

November, 2021

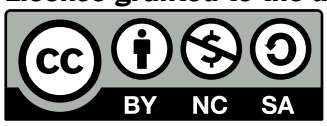
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en

Acknowledgements

I would like to sincerely and gratefully thank everyone who have directly, or indirectly, contributed on this journey:

My Family for their never-ending support, understatement and belief in me, without whom this would have been just another dream;

A special one to my cousin, Patricia Rebelo, for kindly reviewing my Dissertation;

My Teacher, José Carlos Ramalho, for believing and challenging me throughout my journey at the University of Minho;

My Friends and Colleagues for being there to cheer me up when I mostly needed and for inspiring me in many ways to do better;

To ItSector for having me so warmly during such harsh times: Emanuel Pacheco and Isabela Fontoura for their shared knowledge and insightful times during the iOS Academy. A especial one to Helena Brandão and Carlos Bernardino for their guidance and patience throughout this journey.

I am truly grateful for having you all by my side. A huge and warm Thank You from the bottom of my heart!

In honor of my grandfather, Pierre Solans. I miss you dearly!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

V.N. Famalicão, November 7, 2021
(Place) (Date)

Carlos Miguel Rebelo Solans.
(Carlos Miguel Rebelo Solans)

Resumo

Desenvolvimento iOS: Aumentando a Confiabilidade de Aplicações Bancárias com Integração de Mecanismo de Autenticação Forte

Neste momento, as aplicações móveis encontram-se cada vez mais presentes no quotidiano de cada indivíduo, permitindo desempenhar diferentes tarefas, tais como gerir contas bancárias e transação de fundos monetários.

Devido à rápida adoção e desenvolvimento de IoT, é também importante garantir a proteção da segurança dos Utilizadores de ataques cibernauticos, impedindo o acesso destas contas e execução de determinadas operações não autorizadas pelos respetivos títulares de contas via aplicações de Home-Banking.

Tendo isto em consideração, esta Dissertação tem como principal objetivo analisar e integrar uma camada de segurança para Autenticação e Validação de transações de fundos, designada por TrustFactor, numa aplicação existente de Home-Banking.

Visto que a implementação será vocacionada para dispositivos móveis, serão abordados temas relacionados com os Paradigmas de Desenvolvimento de Aplicações e tecnologias usadas no ecossistema Apple.

Palavras-chave: Home-Banking, Desenvolvimento iOS, Desenvolvimento Móvel, Segurança Móvel, CocoaTouch

Abstract

iOS Development: Increasing Home Banking Reliability with Integration of Strong Authentication Mechanism

At the moment, mobile applications are increasingly present in the daily lives of each individual, allowing them to perform different tasks, such as managing bank accounts and transaction of monetary funds.

Due to the rapid adoption and development of IoT, it is also important to guarantee the protection of the users' security from cyber attacks, preventing the access of these accounts and execution of certain operations not authorized by the respective account holders via Home-Banking applications.

With this in mind, this Dissertation's main objective is to analyze and integrate a security layer for the Authentication and Validation of fund transactions, called TrustFactor, in an existing Home-Banking application.

Since the implementation will be aimed at mobile devices, issues related to the Application Development Paradigms and technologies used in the Apple ecosystem will be addressed.

Keywords: Home-Banking, iOS Development, Mobile Development, Mobile Security, CocoaTouch.

Contents

List of Figures

List of Listings

Acronyms

2FA Two-Factor Authentication 5

API Application Programmable Interface 16, 17, 19, 44, 45, 48, 54

CSS Cascading Style Sheet 14, 19

GPS Global Positioning System 30

GUI Graphical User Interface 29, 30

HTML Hyper Text Markup Language 14, 19

HTTP Hypertext Transfer Protocol 35, 36, 44, 46, 48, 50, 51, 52, 55

IDE Integrated Development Environment 30

IoT Internet of Things 1

JSON JavaScript Object Notation 15, 47, 48

MVC Model-View-Controller 30, 39, 49

OTP One Time Password xiii, 5, 6

PIN Personal Identification Number 6, 10

PWA Progressive Web Application 14, 15, 16

REST Representational State Trasfer 2, 37, 46

SDK Software Development Kit 14, 29, 45
SIM Subscriber Identity Module 10
SMS Short Message System 4, 6, 9, 10, 42, 43, 50
SSL Secure Sockets Layer 15

TLS Transport Layer Security 15

UI User Interface 2, 18, 20, 22, 31, 34, 40, 49
UML Unified Modeling Language 13
URL Uniform Resource Locator 35, 47
UX User Experience 22, 31

WWDC World Wide Developers Conference 27, 29, 34

XIB XML Interface Builder 50

1 Introduction

Home-Banking is a term that gained popularity in the 1980s and, by definition, is a system whereby anyone at home, or in an office, may access Bank information's via a computer, with modem connection, and even execute other functions such as fund transactions or service payments and shopping.

This type of service was first introduced by Banks such as Citybank and Chase back in 1981 in New York city. In 1994, the Stanford Federal Credit Union developed the first Home-Banking solution based on a public network, also known as the Internet.

With the rapid growth and evolution of IoT, and the introduction of Smartphones in the market, Bank Institutions have become closer to their clients with the help of Home-Banking Applications. In Europe, after a study conducted by comCore, it is estimated that the adoption of this solutions in mobile devices, such as Smartphones, has increased around 71% in between the years 2018 and 2020 [38].

However, the increasing popularity of these services has led the criminal world's interest in exploiting vulnerabilities on Home-Banking solutions in order to obtain credentials of Bank users accounts. Due to this, Bank Institutions have been putting constant efforts in security research and development of safer authentication and fund transaction validation mechanisms in order to increase the protection of their clients Bank accounts.

With the main purpose of increasing reliability and online security, **Strong** and **Multi-Factor Authentication** mechanisms have been implemented to ensure user integrity throughout fund transactions or during access to certain features.

The integration of Strong Authentication mechanism during fund transaction has been developed in a real-context scenario, on an existing Home-Banking iOS application. This thesis has been written under the context of an internship at the ItSector - a company with vast knowledge and portfolio of solutions targeted to companies from the finance sector, such as Assurance and Bank Institutions.

1.1 Objectives

With this Dissertation we expect to integrate a new Authentication and Transaction Validation using a Multi-Factor mechanism on a Home-Banking Application.

This security layer has been already implemented on the Bank Institution's services. However, in order for it to work as expected, this service needs to be implemented on the client-side with some adjustments on the Applications.

The Application is available for both iOS and Android devices, although our main goal is to integrate the new services on iOS side. Thus, we will provide an in-depth study on the currently used technologies to implement Native Applications.

Even though this Dissertation already has a well defined theme and technologies, we will discuss Paradigms of Mobile Applications, exploring some aspects of Progressive Web and technologies used by Cross-Platform Development Paradigm and how they are applied to iOS Applications.

Therefore, and in a summarized form, the objectives for this Dissertation are established in the following list:

- Understand the Paradigms of Mobile Development;
- iOS Frameworks and Technologies for Native Development;
- Architecture and Design Patterns applied to Mobile Applications;
- Integration of REST Services;
- Strong Authentication and Fund Transaction Validation layer;
- Adapting UI Components

1.2 Document structure

This Dissertation has been divided into two major sections, the State of Art and the Dissertation Core.

Throughout the State of Art, we will discuss about Authentication Mechanisms commonly used by Home-Banking Applications, how they work and their disadvantages. Following that, we will give an overall introduction about a Software based Authentication Mechanism designed especially for online fund transaction based Applications, which is called TrustFactor.

During the Dissertation Core section we will tackle several subjects about iOS and General Application Development. In early chapters we will cover aspects of Mobile Development Paradigms, starting by Progressive Web Applications followed by Cross-Platform. A small general introduction to Native Development has been written, however we will only go in-depth about what it is on the next major section.

The fourth chapter, the **Native Technologies and Frameworks** section, is where we will discuss the technologies used for a Native Development Paradigm. We will start by introducing the **Core Technologies** Apple provides, backed with a background on Objective-C and Swift. Next up we have dedicated a sub-chapter entitled as **User Interfacing Technologies** to introduce both Cocoa/CocoaTouch and

SwiftUI. To close this major chapter, we will also discuss about external libraries during **Frameworks and External Packages**

Following the Native Technologies and Frameworks chapter, we will present the **Development Process**, where we will talk about **Architecture and Design Patterns** used under iOS Development, followed by the **TrustFactor Integration** with the technologies we have agreed to use.

As a form of closure to this project, throughout the **Conclusion** chapter we have made an appreciation of the overall work hereby present, followed by possible **Future Work** under the scope of this Dissertation.

2

State of the art

This dissertation has the main purpose of integrating a Strong Authentication layer on an existing home-banking application. As of today, the App features only one authentication mechanism and two transaction validation mechanisms, triggered under different scenarios, the SMS Tokens and Positions. However, the latter are no longer considered to be strong enough by the bank institution and therefore it has ordered the introduction of a Strong Authentication mechanism with transaction risk analysis features.

That being said, with this dissertation, we will include a transaction validation layer, built by SecuritySide, called the TrustFactor. This fund transaction security layer will be implemented on the iOS and iPadOS version of the Home-Bank App, therefore this master's thesis will also be focused on the available technologies to develop applications for this ecosystem.

In order to justify the usability of TrustFactor we will, hereby, discuss some of the most popular authentication and validation mechanisms used by home-banking solutions.

2.1 Authentication Mechanisms

With the rapid evolution of Technology, the IoT has contributed to the growth of a number devices connected to a network, making it equally more susceptible to malicious attacks, like for example the man-in-the-middle.

As a result of this growth, Cryptography and Security have emerged and evolved with the purpose of protecting private information shared over a network from non-authorized personnel, therefore establishing three important pillars of security - Confidentiality, Integrity and Privacy.

There are different mechanisms that aim to asses the User's integrity, and they fall under two different Authentication Schemes categories: *Simple Authentication* and *Two Factor Authentication*.

Thus, on the sub-chapters that follow, we will discuss about each of these mechanisms, pinpointing each weaknesses and strengths when it comes to Usability and Security.

2.1.1 Simple Authentication

The simplest and more commonly form of authentication and User integrity validation is by using a pair of Username/E-mail and Password. Both of them can be set upon account creation and customized by the User or automatically generated by the Service provider.

Even though this method is the most overused by the majority of online Services, this Authentication Scheme is also more prone to cyber-attacks under public communication channels that are not considered to be safe, like the Internet.

One of the most recurrent User's practices on choosing Passwords for Online Services is to create one short and easy to memorize, but these are more prone to theft or guess, while lengthy and complex Passwords are the opposite way. However, as length and complexity increases, the probability to forget them also increases [35].

Moreover, independently of how big and strong the Password is, by combining letters, symbols and numbers, in order to overcome the problem of forgetting it, Users have a tendency of re-using the same Password and not changing it very frequently. Another major problem is that upon any account creation, the User is completely unaware on how it will be stored or where, and the more they use a given Password the less safer and stronger it is, because if they are stored without any encryption or if any data breach happens, the more likely an attacker can access other Services like E-mail, Social Networks and even Home-Banking platforms.

2.1.2 Two Factor Authentication

The exponential growth of networks has resulted on a high number of devices connected to the World Wide Web and, as a consequence, many Services have risen with protected resources. Due to this phenomenon, as we previously exposed, a Simple Authentication mechanism on its own is no longer considered to be secure enough.

To address this issue and to enforce a better security on Simple Authentication mechanisms, many Service providers have come up with a different set of rules which forces their users to change their Passwords every once in a while and, in other cases, as we have previously stated, these Passwords must match certain rules like mixing Symbols, usage of capitalized letters and numbers. However, this has a huge negative impact on Online platforms Usability [39].

Thus, with the purpose of addressing these issues, a new type of Authentication Scheme has emerged, to verify the User's integrity, entitled as Two Factor Authentication - also commonly referred as 2FA.

There are many implementations of a Two Factor Authentication mechanisms, but under the hood, to authenticate users, they are based upon One Time Passwords, commonly know by the acronym OTP.

Nowadays, OTP Authentication Schemes, can be found on a wide variety of Online Services, whether on Social Networks, E-mail providers and especially on Bank Accounts in order to validate User's Integrity when executing fund transactions.

As its name indicates, Authentication Schemes based on OTP work with Passwords, or Tokens, can only be used once and are automatically expired when the Transaction or Authentication has been executed or time to live has passed its due.

Thus, Authentication and User Validation Schemes based on OTP have three main characteristics which should be respected: eligibility for just one attempt, only a valid login and only for a short fraction of time [39].

The majority of Two Factor Authentication schemes implementations are based on three additional characteristics, like **Knowledge** - for example, a pre-defined Password or PIN known by the User -, **Possession** - something the User has in order to proceed with Authentication, like a Matrix Card or Authorization Device - and at last something **Unique** to each User - biometric information like fingertip, facial or voice recognition [30].

In Home-Banking Applications Two Factor Authentication schemes such Matrix Cards and SMS stand out compared to other existing solutions. Besides these, after a survey we have conducted about Online Experience and Behavior, answered by 134 individuals, out of each 115 are Home-Bank Users, we learned some Bank Institutions even use E-mail Services to Authorize and Execute Home-Bank fund transactions.

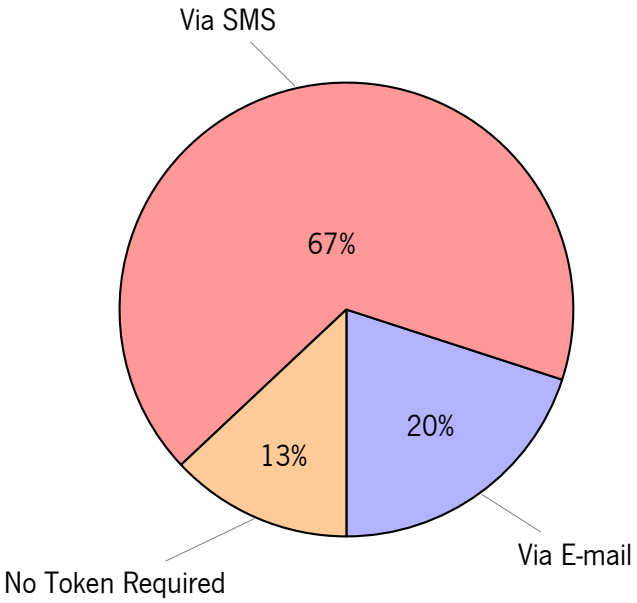


Figure 1: Authentication Schemes Used as OTP in Bank Transactions

2.1.2.1 Grid or Matrix Validation Cards

The oldest and most primitive mechanism to validate User's integrity under OTP schemes is to provide a list of previously randomly generated characters or numbers, like Matrix Cards. According to the survey we have conducted, this mechanism is still used by 67.5% out of 115 Home-Banking Users.

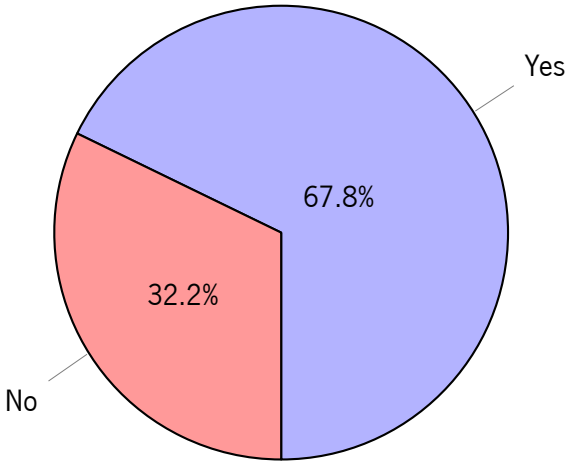


Figure 2: Usage of Matrix Card in Home-Bank Transactions

Matrix Cards can be represented by different type of digits, numbers or letters. As an example, we have designed a sample Matrix Card, depicted by 3, following a numerical format with three randomly generated digits.

	1	2	3	4	5	6	7	8
A	123	180	555	554	889	804	0471	830
B	321	240	646	444	634	999	0013	265
C	231	0001	666	52	654	464	343	0050
D	134	832	0010	333	546	472	777	484
E	121	929	882	808	263	753	783	500
F	122	0099	942	810	209	604	073	204
G	430	142	911	200	100	670	181	201
H	120	433	921	311	801	154	0155	217

Figure 3: Grid Card Example

The way these Authentication Schemes work on Home-Banking solutions are quite simple: most of the times, when the User requests a fund transaction, the Application prompts one or more digits accordingly to a coordinate in the grid and, in some cases, digit index can also enter the equation. As an example for the grid card shown in Figure 3, the Application could request a number on second index with a coordinate C5 - this corresponds to the number 5.

Even though this is a straightforward, simple and widely used Authentication Scheme, the Grid Cards are a physical object susceptible to loss or even theft.

With the fast climb of internet usage, Matrix Cards are also threatened by Phishing and unfortunately, in some cases, Online Bank services can't do much about it to prevent these attacks other than make their users aware of this danger. One good example of this problem can be depicted by Figure 4, where another platform, that looks pretty much alike the official Home-Bank Website, requesting someone to enter every single existing number on the Matrix Card.

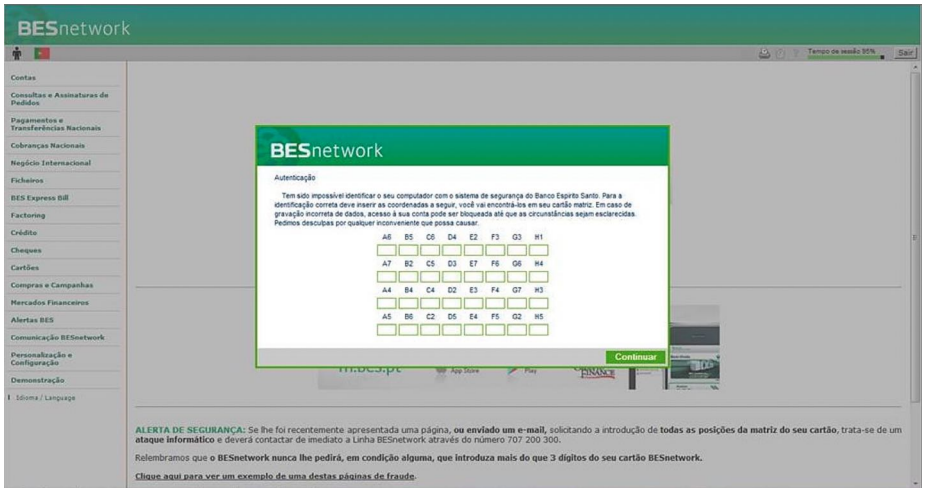


Figure 4: Attempt to obtain Grid Card information through Phishing

2.1.2.2 Hardware Tokens

Just like Grid Cards, Hardware Tokens are the oldest existing Authentication Schemes, however they are not as frequently used by today's Home-Banking solutions as a result of the ascension from other Authentication mechanisms.

The Hardware Tokens, depicted by Figure 5, are small devices which looks quite similar to the format of a USB Pen Drive or a simple Credit Card. These devices also features an internal memory, though limited, where one or more Tokens are stored for being used later throughout Authentication Services or Applications supporting them.

Figure 5: Hardware Token

There is a wide variety of Hardware Tokens brands and different models with distinct characteristics: some may feature keypads to input Passwords, biometric readers, wireless antennas, among other features aiming to maximize their security [3]

Even though these devices provide an increased level of security to Services and Applications, at the end of day it is a physical device that can get easily lost or stolen, just like Grid Cards. Hardware Token Devices features also have limitations and problems, like their expensive price tags, expiration dates and a battery that once drained can't be recharged forcing Users to acquire a replacement device.

2.1.2.3 SMS Validation

As observed by our survey results, depicted by Figure 1, the most common Authentication mechanism used by Home-Bank applications are based on SMS, or Short Message System.

In Bank solutions, to take advantage of this mechanism, their Clients must provide a correct and trusted cellphone number when the Bank Account is created. Whenever there is a Transaction attempt the Bank Services send a text message with a generated Token to execute the transaction.

Under a User Experience point of view, this is the best and most useful scheme for the Users, considering it eliminates the need of requesting a Grid Card or Hardware Token Devices. On top of that, in iOS, Apple offers a possibility to automatically input the Token received by SMS for the User, if the transaction is being made on the same device with the destination cellphone number, as depicted by Figure 6.

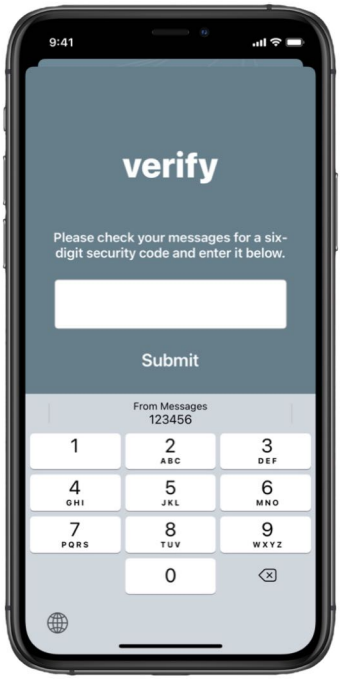


Figure 6: Auto-fill SMS-delivered Security Token

However, the integration of this mechanism is prone to Security threats. According to [9], unauthorized personnel or cyber criminal individuals can steal/extract data and even deceive Users who have subscribed the Authentication scheme with Phishing, since many cellphone carriers allow their clients to receive messages coming from different public Websites or Services, which are not considered to be safe, who are able to pretend being the Bank Institution.

There are many different tactics or methods available to exploit vulnerabilities in text messages sent via SMS under the context of Home-Bank transactions, among which stands out:

- **Phishing**: like previously mentioned, a malicious entity tries to claim being the Bank Institution by sending links referencing Services able to extract data;
- **SMS Sniffing**: for example, by cloning a SIM card, individuals are able to intercept messages containing Tokens or PIN since these are not encrypted in transit;
- **Malware** or **Spyware**: by installing third-party Hardware or Software able to steal data stored on device by sending over a Network;
- **Wrong Number**: sometimes **Tokens** could be sent for the wrong person, this can happen when one dials the wrong number or the person has gotten a new one;

All the problems we have hereby exposed can have a huge negative impact to one organization, compromising their the Client's trust, public image and financial consequences to both parties.

2.2 TrustFactor

TrustFactor is an Authentication and Transaction protection Service that allows Web Applications to protect sensitive data and operations from being executed by unauthorized personnel, like for example Service Authentication, changing Password or other account's information and Fund Transaction approval on Home-Banking solutions, by providing a Strong Authentication mechanism that aims to protect Clients from cyber-security threats like *Phishing* or others.

This Service is structured in three application layers with distinct purposes that cooperate with each other for the safety of users, which are:

- **TrustFactor Stack**: is the whole set of Services which ensures the operation and integration of the Home-Banking solution with TrustFactor;
- **TrustFactor Cloud Services**: a responsible layer for communication and validation of the generated Authentication codes;
- **TrustFactor Agent**: is a free Application available for both Android and iOS devices to be used by the Bank's Clients

This provides a mechanism that allows Users to authorize different transactions in the Home-Bank, having full knowledge of their parameters and a Risk Analysis of their execution that can be classified as Low, Medium and High. The classification can be based on several parameters such as the frequency of transaction made to a certain destination, the localization from where it's been requested and how much money will be sent, as depicted by the Figure 8.

2.2.1 TrustFactor Configuration

The TrustFactor Authentication Service is passive to different configurations, whether Cloud based or on Bank Institution's own service. This way, the Bank can opt out by using the Service completely allocated on SecuritySide's Services or Internally and have full control on the platform.

The diagram depicted Figure 7 displays one possible configuration where TrustFactor has been fully integrated on the Bank's Internal Services. By choosing this configuration the Bank Institution is also able to customize the TrustFactor Agent Application and distribute it as their own.

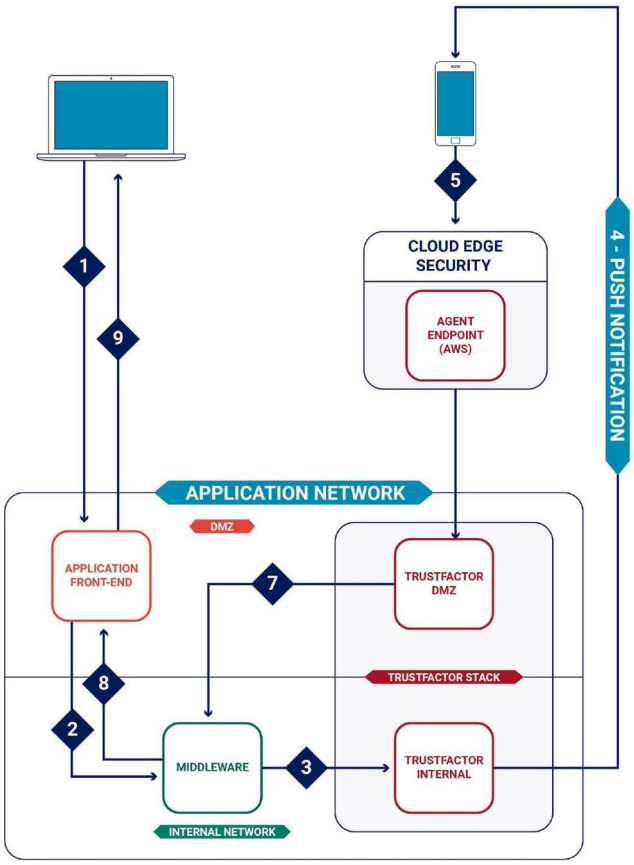


Figure 7: TrustFactor Overall Configuration

Besides the Architectural configuration, the Diagram also depicts the overall operation steps of TrustFactor, where a Client (**1**) accesses the Frontend Application and requests a Transaction (**2**). The Transaction request is received on the Backend Services through a Middleware which verifies the Authentication mechanism set for transaction. If the configured mechanism is set to be TrustFactor then the request is forwarded to TrustFactor Authentication Services (**3**) that should fire a Push notification on the device with associated contract on TrustFactor Agent Application. The User can use the Application to approve or reject the transaction - we will cover this step on future chapters.

2.2.2 TrustFactor Agent Operation Flow

As we have mentioned earlier when we first introduced the TrustFactor Service's layers, the Agent is an Application available for both Android and iOS, completely free for the Bank Institution's Client.

This Application works like a cryptographic Software Token on supported Home-Banking Applications. The User can authorize one or more Agents for Transaction approval and Authentication.

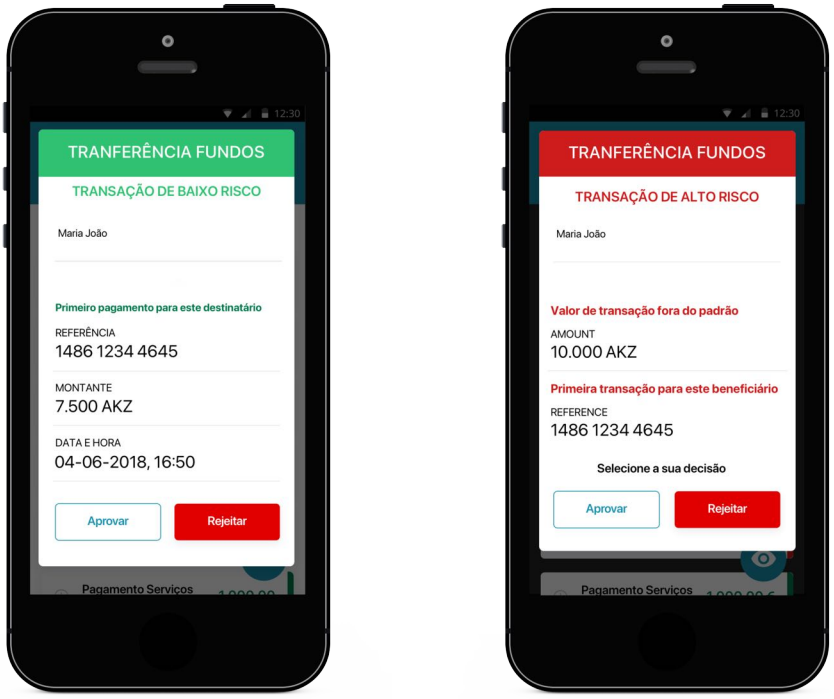


Figure 8: Example Fund Transaction Approval with Risk Analysis

Once the Validation mechanism is integrated in the App and after the Home-Bank Client has successfully associated their contract with TrustFactor, every time a Transaction that uses this Authentication mechanism is requested a Push Notification is sent to the associated Mobile Devices

In order to proceed with the Transaction, on iOS the User should open TrustFactor and, as depicted by Figure 8, the Agent will display every Transaction details for approval, such as Destination Reference,

amount, date-time and, in some cases, other properties such as location where the Transaction has been requested. These parameters will be used to asses the level of Risk associated with the execution of Transaction.

The aforementioned steps have been translated to a Sequence UML Diagram depicted by Figure 9 for a better understanding.

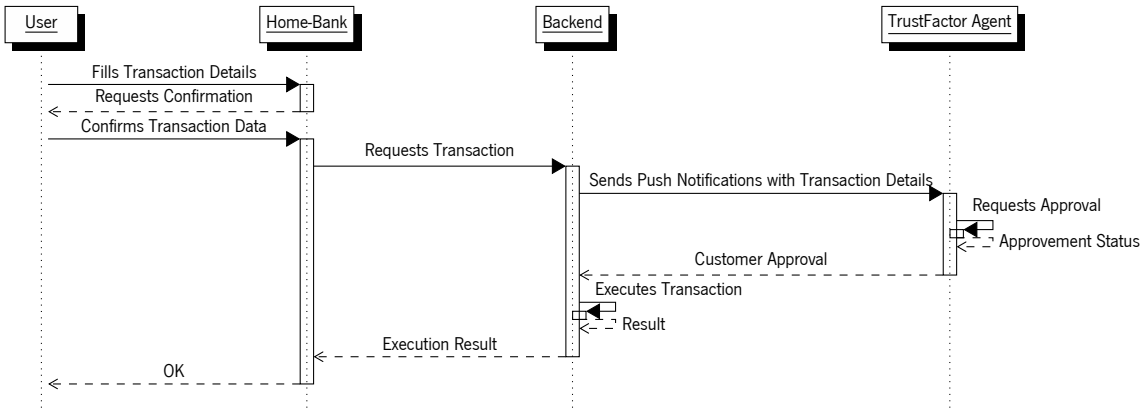


Figure 9: Fund Transaction Approval Flow

2.3 Summary

Within the State of Art chapter, we started by exposing the main goal of this Dissertation, which is integrating a TrustFactor authentication layer on the client side Application. With that in mind, we started by revising the existing types of Authentication Mechanisms, which can be divided into the categories of Simple and Two Factor Authentication.

Since TrustFactor falls within the Two Factor Authentication and has been conceived to Home Banking, we have focused ourselves on existing authentication solutions used within these platforms, pinpointing their differences, advantages and also as their disadvantages or vulnerabilities.

To close this chapter, we have also introduced TrustFactor as an authentication service and an overview of how it can be applied and used under a Home-Bank platform.

Since this implementation will be done on the iOS Application, our goal for the chapter that follows is to discuss about Mobile Development Paradigms and document how iOS Applications can be built.

3

Mobile Development Paradigms

When the first iPhone was announced, back to January 2007, it was a simple mobile phone with a few extra features, such as a media player for music, videos and images, the possibility of surfing the web through Apple's flagship Software called Safari and the ability to fetch and send E-mails, on 3.5 inches multi-touch screen.

A year after the launch of this device that revolutionized the industry of mobile computing, Apple recognizing its potential and in order to extend its features, introduced the App Store and a set of SDK tools so developers could create Applications and distribute them to potential users via this service.

As time went by, Software Development has been innovating and new technologies emerged, such as new methods and Application Development Paradigms, making the development process easier and less time consuming, regardless of its target environment.

On the following chapters, we will discuss these Mobile Application Development Paradigms - such as Progressive Web Applications, Cross-Platforms Applications and, at last, Native Applications - pinpointing their advantages and disadvantages when it comes to iOS and iPadOS development.

3.1 Progressive Web Application

Progressive Web Development, which is also known by the abbreviation PWA, is a mobile development paradigm which distinguishes itself from the others due to the fact of their simplicity and having the capability of being accessed by any type of devices, regardless of the Operating System, via a Web Browser or a WebView embedded on a Native Application.

That being said, PWA are considered small applications based on Web Technologies, combining a markup language, such as HTML, to structure the skeleton and represent information, a Cascading Stylesheet (also known as CSS) for designing purposes and a programming language like JavaScript - or as in more modern frameworks such as React, Vue or Angular, its super-set called TypeScript. In the Figure 10 we display an example on how a News Progressive Web Application looks like through the Browser, both on the iPhone (on the left) and on Android (on the right).

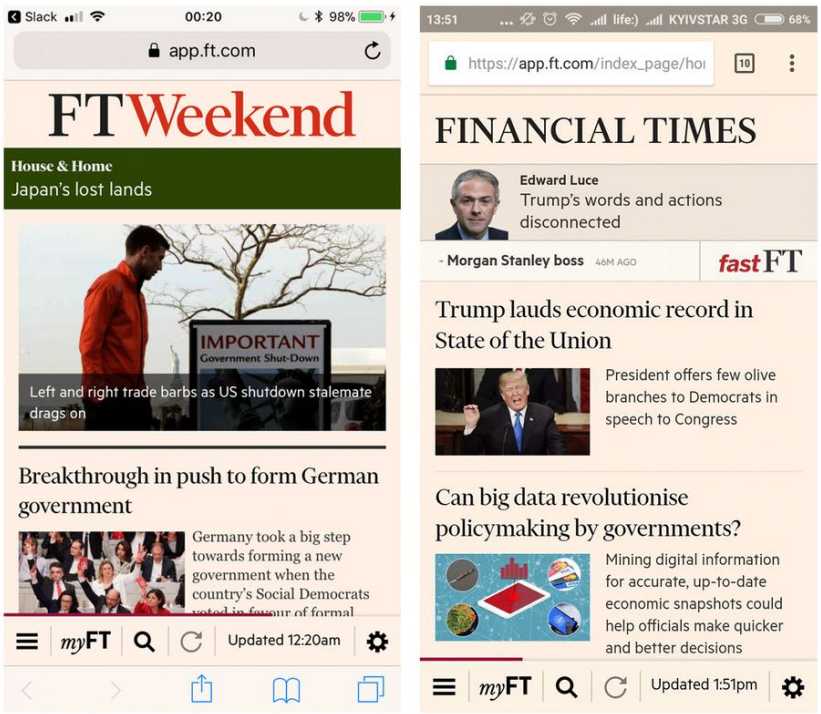


Figure 10: Progressive Web Application Example

In the early days, this type of application proved to be quite limited on mobile devices by lacking simple features Native Applications had to offer, such as the ability to receive Push notifications, save content for offline access and, for example, when it comes to iOS users, they were not able to save the Application on the Home-Screen for a eventual quick access. However, some of these issues have been addressed with the release of new major Operating Systems [34].

In spite of these Applications being built upon Web technologies, not all Web Sites that have the features described above can be considered PWA. For this to happen, these Web based Application must support the following key features:

- **Secure Contexts**: the Web-based App must be distributed to the end-client over a secure network, assured by protocols such as TLS or SSL. In some cases, PWA features such as geolocation and a few Service Workers will only be available by assuring this feature;

- **Service Workers**: a script that performs network interceptions and Browser cache control. By using Service Workers one can, for example, store content for a offline access;

- **Manifest**: it's a JSON configuration file which specifies how an App should be presented to the end-user and allows it to be discoverable over the Web. In the following code excerpt we can see some of the most common properties of a Manifest configuration file.

```
{
    "name": " HomeBanking ",
    "short_name": "HB",
    "start_url": "/?source=pwa",
    "icons": [{
        "src": "/images/logo.png",
        "type": "image/png",
        "sizes": "192x192"
    }],
    "theme_color": "#28527a",
    "display": "fullscreen", // controls the Browser UI by hiding certain components
    "orientation": "portrait" // or landscape
}
```

Listing 1: Manifest Configuration Properties

As it has been stated, the limitation to certain PWA features have only been addressed in 2018, with the release of iOS 11.3, allowing features such as geolocation based on iPhone API, access to certain sensors such as Magnetometer, Accelerometer and Gyroscope, camera and audio output as also access to their own payment method, the Apple Pay, limiting other features for Native Applications.

The more interesting features that have been limited on iOS by this type of Applications are the ability to store content for offline access, which is currently limited by a maximum of 50Mb storage, and the ability to receive Push Notifications. Security-wise, some API have been strictly limited to Native Applications such as Touch ID and Face ID [34].

To sum up and despite all the limitations exposed, the fact PWA benefits from Web technologies, it becomes easier to develop apps compared to Native solutions. With PWA running on the Browser, it's also easy to develop an app targeting multiple devices rather than iOS all alone, making it cheaper and easier to maintain. However, if we want to develop an App that relies a lot of Hardware or other Operating Systems core features - such as Push Notifications - and build great User Experience [37], it might be wise to reject this development paradigm.

3.2 Cross-Platform Application

Applications developed via Cross-Platform paradigm are easier to develop than Native Applications and because most of their code can be shared across devices [19] with different Operating Systems.

Another good reason why a company should make the switch to a Cross-Platform Development is the cost associated with it when it comes to required human resources for developing and maintaining the Application, since there is no need in developing two standalone Applications targeted to devices with different Operating Systems. Moreover, the effort in Quality Assurance - also known as Testing - can also be reduced since there will be a single Application to test [28].

Despite the high potentials of the Cross-Platform Development Paradigm and reduced costs of development, there are a few drawbacks associated with this development paradigm.

To name a few, some of available frameworks and technologies have yet limited access to certain device API and the User Interface might not be compliant across other Operating System components. Another important aspect that should be taken into account when developing an app is that third-party frameworks are not free of bugs, there may be hardware compatibility issues as performance glitches that can definitely impact the Application User Experience [27].

Even though Cross-Platform Development tries to eliminate the need of writing code in different languages and technologies, in order for these applications to run on different Operating Systems, the framework acts as a layer on top of Native Technologies foundations. For example, User Interfaces are implemented as a Web Page [14], although it is presented to the end-user via a Native Application by relying on the WebView component.

In order to better introduce these technologies, we have selected some of the most used frameworks, according to a study conducted by Stackoverflow in 2019 about the "Most Popular Technologies"[1] among Professional Developers, and crossed information on their respective documentation and articles. On the following sub-chapters, we will discuss about React Native, Cordova and Flutter.

3.2.1 React Native

React Native is an open-source framework created and maintained by Facebook, that aims to combine the best parts of native development with core React features [33] and therefore reduce the costs of development. Even though the framework is built and more commonly used via JavaScript, it is also flexible enough to be integrated on existing applications built with Java, Objective-C or Swift.

This is a widely used framework in the industry by Applications such as Facebook, Instagram, Netflix, The New York Times and many other companies [2]. Also among Professional Developers, accordingly to Stackoverflow, it is used by 10.8% out of the 49861 developers who have answered the survey.

Since React Native is Cross-Platform framework, in order for an Application built upon this framework to run, still needs some foundation layer of officially supported technologies. Therefore, the Figure 11 depicts the core architecture used in React Native, where two bridges on each side act as a communication layer between native and JavaScript components [36].

[1] Most Popular Technologies: https://insights.stackoverflow.com/survey/2019#most-popular-technologies
[2] 10 Famous Apps Using ReactJS Nowadays: https://brainhub.eu/blog/10-famous-apps-using-reactjs-nowadays/

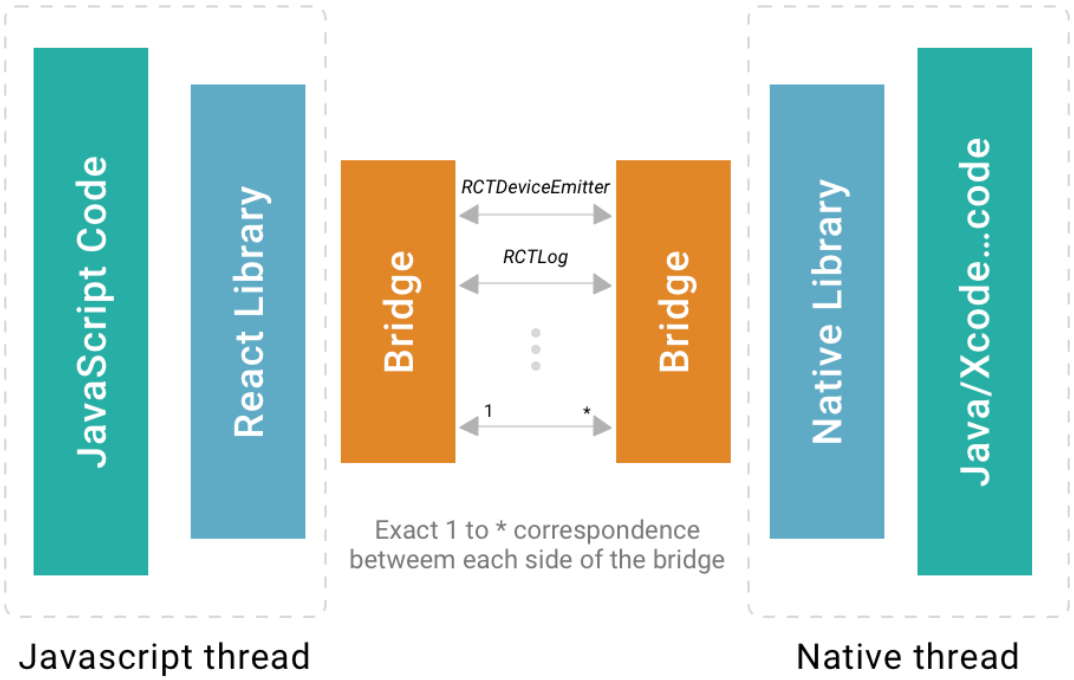


Figure 11: React Native Architecture [36]

Choosing React Native has many advantages, such as speed of development for multiple platforms with shareable code among them, a simplified UI and a large developer community. By using this framework team sizes can also be reduced, having no impact whatsoever on low budget projects.

Although one should take into consideration that React Native is a new technology which is still under development, so compatibility issues are still being found and updates being released frequently to patch them. It also lacks on basic features such as push notifications [11] requiring developers to implement them using officially supported Native technologies.

3.2.2 Cordova

This framework was conceived by several engineers from Nitobi, a company dedicated to the Web Development, during an event about iPhone development hosted by Adobe in San Francisco [17], shortly after the introduction of the second version iPhone in 2008.

A Technology called UIWebView which allowed developers to load Web resources onto a native application has been explored with the purpose to investigate how an Web Application, or Progressive Web Application, could run natively without putting extra effort on learning the Native Technologies. This experiment has been successfully accomplished and gave origin to a project called PhoneGap which, later in

2011, was bought by Adobe who has donated its framework to Apache Foundation. The latter has then decided to rename the project to Cordova.

Right now, this is an open-source framework that follows the original principles, however it is now supported by a wide-range of devices running different Operating Systems - iPhone, Android and Windows Phone. This framework also allows modern web development technologies, such as HTML5, CSS3 and JavaScript, for a Cross-Platform development.

The Architecture of this frameworks remains loyal to its original, where the User Interface components are implemented by a WebView and a layer of Cordova Plugins which now allows to interact with a set of API and Native components - such as battery, camera, geolocation services, file management, among others. The Figure 12 gives a representation of Cordova's Core Architecture.

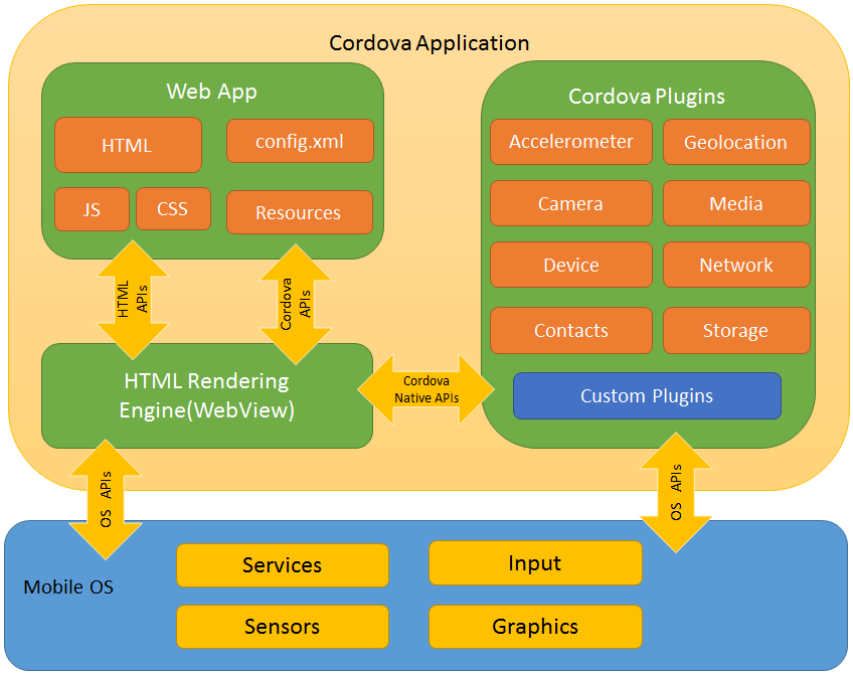


Figure 12: Cordova's Core Architecture [13]

Accordingly to the official framework documentation, its usage is highly recommended when developers need to deploy an existing Web resources (as for example a fully operational Progressive Web Application) to be distributed on a Native Application [14], or when there is the need to create a mixture between Native Application components with WebView components.

However, and despite the maturity and wide existence of plugins, Cordova also features some limitations due to the technology it is based on, the WebView, which compared to other technologies, and native components, has a slower performance and it can be further aggravated if the application includes graphics and some animations. Another major downside important to mention is that just like other non-native open-source frameworks, documentation might lack in quality and plugins might have bugs which affect correct behavior across different versions of the Operating Systems [16]

3.2.3 Xamarin

Xamarin is one of the oldest and most used frameworks on Cross-Platform Development. This framework has started as a project from the self named company back in 2011, and was recently bought by Microsoft in 2016 who merged it on the services pack shipped with Visual Studio [18].

Applications that are developed using Xamarin, in contrast of the other technologies we have mentioned before, do not rely on Web technologies to be implemented. Thus, Xamarin uses Microsoft's own programming language C# and XAML Markup Language. By using these technologies, Microsoft emphasizes on its documentation that developers are able, in most cases, to share 90% of the written business logic and 80% of UI Components across other platforms [26] in the market - Android and iOS. The Figure 13, obtained from Xamarin's official documentation, Microsoft represents how the high level architecture communicates with the respective supported native technologies, Objective-C (iOS) and Java (Android).

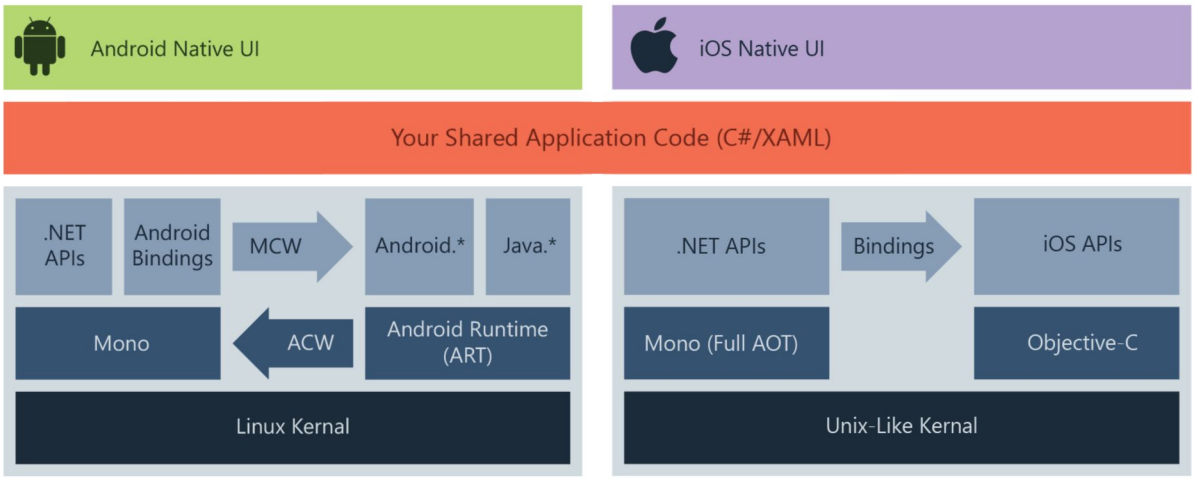


Figure 13: Xamarin's Core Architecture [26]

Besides the great amount of shareable code, another great advantage for using Xamarin for Cross-Platform Development, is the fact it supports a wide range of Operating System versions, starting from iOS 9 [10], when most of Cross-Platform technologies require iOS 11 or later.

This technology seems like a great candidate for Cross-Platform development, however Microsoft has announced it has a few technological limitations imposed by iOS architecture and native programming language [31]. Besides these limitations, Microsoft has also claimed during an event that Xamarin will become deprecated and a new development framework will be introduced on .NET as a replacement called .NET Multi-Platform APP UI [2], which might eventually require Software Developers to transition their apps to this new framework.

At last, drawbacks such as Xamarin being charged for Professional and Enterprise usage, limited access to open-source libraries and the required size for simple applications, there may be enough reasons for a developer discard this framework on certain projects.

3.2.4 Flutter

Flutter is one of the youngest framework on its market. It's also an open-sourced technology, first launched in 2018, introduced and maintained by Google.

Just like other Cross-Platform frameworks, Flutter has been designed to allow the maximum re-usage of code between different Operating Systems and also Browsers. In addition to previous studied technologies, it also allows to build applications capable of running natively in Desktop environments, such as Windows, macOS and even Linux.

This framework uses Dart as a programming language which is later compiled and translated to natively supported languages. Despite the need to compile the App to run it, throughout the development phase these are executed upon a Virtual Machine which allows to apply code changes instantly - a technology that is also known as *Hot Reloading*.

Internally, Flutter has been conceived inspired on React, featuring Reactive Interface capabilities. In other words, every user input that impacts internal state data is automatically reflected on the components that depends on them, without having the need for a developer implement himself such control mechanism.

In its Architecture, represented by the Figure 14, in order to support different environments, Fluter has been split into three layers with distinct purposes [15]:

- **Framework Dart**: is the highest layer on its hierarchy, where developers implement the application by using Dart. This layer holds Foundation classes that allow developers to specify animations, gestures and the layout; a Widget Layer that through composition make up the user interface and at last Material and Cupertino serves as UI primitives for Android and iOS, respectively;

- **Engine**: is a lower level architecture, on which the entire framework is based on. This layer has been conceived with C and C++, having the main purpose of communicating with primitive functions which allow, for example, network calls, scene rendering, graphics and input and output file management;

- **Embedder**: is the lowest level layer on the hierarchy, that acts as a translation layer to native technologies - Objective-C in iOS and macOS, Java and C++ for Android and C and C++ for Windows and Linux platforms.

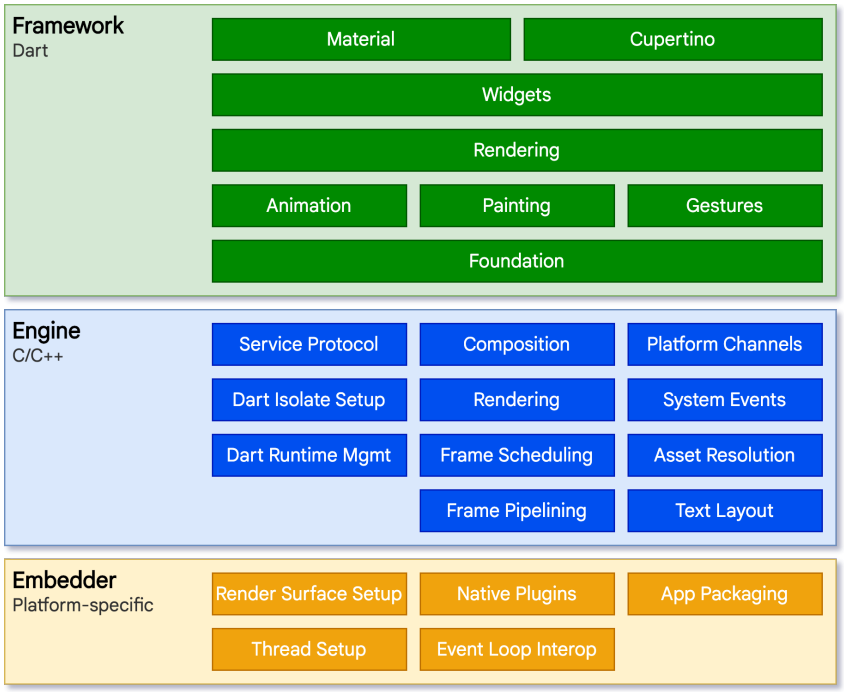


Figure 14: Flutter's Core Architecture [15]

Like any framework, Flutter also has its limitations and drawbacks. The first to be noticed, according to its documentation, is the fact it does not ship with an official debugging tool to ease the process of development. To address this, Flutter recommends developers to use non-official existing tools in order to overcome this limitation.

On Android, Flutter's engine has a minimum variable weight between 4.3Mb on ARM and a minimum of 4.6Mb on ARM64 platforms. On iOS, however, a simple application reaches a minimum value of 10.9Mb, accordingly to their documentation Frequently Asked Questions page[3], due to an encryption mechanism on the Apple's side, making the IPA compression algorithm less efficient. Taking this into consideration, as the Application grows, there might be a considerable impact on its final size.

3.3 Native Application

Native Applications are tailored to a specific mobile platform, thus developers who choose this type of Development Paradigm are required to use natively supported technologies.

This Paradigm of Development has many advantages such as great support to Hardware technologies and full API access, while other technologies may be limited. According to some reports, Native Development is also a great way to developers build Intuitive UI/UX design that follow specific guidelines from

[3]Flutter FAQ page: https://flutter.dev/docs/resources/faq

the device manufacturers. Unlike other Development Paradigms, Native Technologies does not depend on translation layers and therefore developers are able to build apps with better performance.

However, since every Operating Systems supports different technologies, this Development Paradigm can be a drawback as it requires to build the same Application logic but with different technologies in order to support, for example Android and iOS, since the first supports Java and the latter Objective-C or Swift. Therefore, this also has an impact on the required human resources which also demands higher development costs.

3.4 Summary

In this section, we have briefly introduced the existing paradigms of mobile development and supported types of Applications on iOS and iPadOS devices.

Starting from the Progressive Web Application, where it's mainly used Web Technologies to enhance websites to be supported by any type of mobile devices. We also have pointed it's limitations when it comes to running on iOS and iPadOS devices, discarding this paradigm of development by the lack of Push Notifications, access to the device's Hardware capabilities and the Application limitation sizes.

Furthermore, it has been introduced the Cross-Platform Application Development Paradigm, where we have discussed some of the most used frameworks to achieve Cross-Platform Development, the advantages of this paradigm but, as usual, the disadvantages that comes with it.

For a full section closure, we have also introduced Native Application Development Paradigm, the advantages and drawbacks and why would one prefer this type of Application Development Paradigm over the previous ones we have discussed.

4

Native Technologies and Frameworks

4.1 Core Technologies

As we have previously stated, the Native Development Paradigm forces developers to use officially supported technologies.

In order to develop Applications for the Apple ecosystem, developers must either use Objective-C or Swift. In some cases, both can co-exist on a single Application.

On the following sub-chapters, we will introduce some background about each languages, pinpointing each other's differences.

4.1.1 Objective-C

Starting with Objective-C, also commonly known as ObjC, is a Programming Language created in the 1980s by Brad Cox as an extension to C language functionalities [40].

In 1988, this language has been licensed to and introduced initially in the market on NeXT Computers, which later has been acquired by Apple. As a result of this acquisition, Apple began using NeXT Computer's components as a foundation for its own Operating System for Mac Computers called macOS[1], which also includes the Objective-C language. Since Apple's mobile Operating System is based on the Mac technologies as its foundation, Objective-C is also the primary language used for Application's development.

Apple considers this language as a C superset [22], having extra capabilities such the Object Oriented characteristics - Encapsulation, Inheritance, Polymorphism and Abstraction.

Since Objective-C is built upon C, it supports the same primitives data types. However, Objective-C also deals with data types as objects and by doing so provides developers functions to deal with data more efficiently. The foundations of Objective-C provides the following data-types:

- **NSString**: used to store text or a set characters;

[1] Formerly known as OS X

- **NSNumber**: allows to store different types of numbers, like Integers and Floats;
- **NSArray**, **NSSet** and **NSDictionary**: stores a set of data types upon creation;
- **NSMutableArray**, **NSMutableSet** and **NSMutableDictionary**: stores a set of data types and allows it to be modifiable after creation;
- **NSValue**: represents other data structures supported by C language

Since this language derives from C it allows backward compatibility. For this to happen, every implementation requires a signature header. To represent some of these concepts, we have written a class which defines a person, depicted by Listing 2 to 3.

On a first sight, the syntax differences are quite notorious compared to the C language.

To define a class where we want to take advantage of Objective-C data types, it is important to import the Foundation header. Following to that, the class must inherit the **NSObject** implemented by the framework.

The property tag allows us to define the variables *name* and *birthYear* and automatically defines their getters and setters. Next, we must declare all the implemented functions to be defined as public - in this case, we declare a constructor *initWithName* and *displayAge* function.

```
// Header File: Person.h
#import <Foundation/Foundation.h>
@interface Person : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSNumber *birthYear;

// Public Functions
-(instancetype)initWithName:(NSString*)name birthYear:(NSNumber*)year;
-(void)displayAge;

@end
```

Listing 2: Person Class Header File

Just like it happens with C, a header file is not enough by itself, since it does not specify function's behaviour, just their definition. This should be implemented on a file followed by a suffix file-type *m*. On the example 3, we define a simple class constructor and a method which interacts with C language primitive data types with Objective-C's.

```
// Implementation File: Person.m
#import "Person.h"
@implementation Person

-(instancetype)initWithName:(NSString*)name birthYear:(NSNumber*)year {
  self = [super init];
  if(self) {
    self.name = name;
    self.birthYear = year;
  }
  return self;
}

-(void)displayAge {
  int age = 2021 - [self.birthYear integerValue];
  NSLog(@"%@ has %d years old", self.name, age);
}

@end
```

Listing 3: Person Class Implementation File

At last, to use classes instance methods in Objective-C, we need to allocate them to the device's memory. To do such, we call the function's name *alloc* followed by our initializer, which therefore allows us to invoke any given function. In our case, we invoked *displayAge* which should print out "Adam has 23 years old"on the console.

```
Person *person = [[Person alloc] initWithName:@"Adam" birthYear:@1998];
[person displayAge];
```

Listing 4: Executing Classes Code Blocks

The Foundation framework does not only have a definition of data structures. In fact, this framework features a wide variety of essential functionalities that act as a base layer for any given applications, such as interfaces for services like Handoff[2], notification management system - which will be explained throughout Development Process -, deals with errors and exceptions, file management and also networking features. We will not go into much detail right now as we will tackle some of these technologies later on the Development Process chapter.

4.1.2 Swift

The Objective-C has been around since the 1980 with none or very little improvements over the years. It has a complex syntax and according to some developers it gets hard for projects to scale. To address these issues, Apple has decided to develop a new programming language with some Objective-C features, but by eliminating the C language dependency.

[2]Direct Communication across devices logged to the same Apple ID and Network

Swift is a general-purpose programming language announced by Apple during the WWDC back in 2014, built on a modern approach and it's highly characterized by the properties of Safety, Fast and Expressive.

Since it's a general-purpose programming language, Swift is also open-source and flexible enough to be used for both Application development, mobile or desktop, and on large-scale cloud services [23] with the help of frameworks like VAPOR.

Back to its properties, Swift is said to be Safe because it eliminates existing C and Objective-C classes and code which are not considered to be safe, variables must be declared before usage and memory is automatically managed which prevents eventual memory leaks.

Other important aspect that guarantees the Safe property is the fact that in Swift objects cannot be used when their values are null - or nil - and its usage is considered to be a compilation error. However, in some cases, it is hard to make sure a specific value isn't null. To address this, Swift has a validation mechanism called *optionals* which can be used by adding a question-mark at the end of the variable, as depicted by the code excerpt in Listing 5.

```
myDelegate?.scrollViewDidScroll?(myScrollView)
```

Listing 5: Prevent Nil Functions or Variables from being accessed with Optionals

In Objective-C, such mechanism and null values control doesn't exist and can lead to Software malfunction if not controlled beforehand by the developer. As a contrast between Swift and Objective-C, the code excerpt in Listing 6 depicts one approach to tackle with this problem.

```
if (myDelegate != nil) {
    if ([myDelegate respondsToSelector: @selector(scrollViewDidScroll:)]) {
        [myDelegate scrollViewDidScroll: myScrollView]
    }
}
```

Listing 6: Example of Optional Implementation Approach in Objective-C

Another advantage for using Swift is the fact that because of its syntax simplicity it has a low steep learning curve for beginners and Apple has a great documentation support to cope with it, along with free videos from coding events directly shot from the WWDC.

Beyond its syntax simplification, and besides the existence of Classes for the purpose, Swift has also introduced Structs for Object creation. Although both have the same goal and some common features, such as define properties, initializers and methods, Classes have access to extended capabilities such as inheritance, type casting and de-initializers. Another difference between the two one must take into account, is that copies of Classes are passed by references - meaning that changing the original object properties will reflect on its copy - while Structs copied by Value meaning that internal changes will not reflect on existing copies. To serve as a contrast between on Objective-C and Swift Object creation and to

exemplify these concepts, the Listing 7 depicts property mutation of original Struct having no impact on its copy while the Listing 8 depicts property mutation of Classes being propagated.

```
struct PersonStruct {
    var name: String;
    var birthYear: Int;

    init(name: String, birthYear: Int) {
        self.name = name;
        self.birthYear = birthYear;
    }

    func displayAge() {
        let age: Int = 2021 - self.birthYear;
        print("\(self.name) has \(age)");
    }
}

var personWithStruct = PersonStruct(name: "Adam", birthYear: 1998);
var personWithStructCopy = personWithStruct;
personWithStruct.birthYear = 2000;

// Will print `Adam has 21`
personWithStruct.displayAge()
// Adam has 23
personWithStructCopy.displayAge()
```

Listing 7: Struct Implementation and Copies

```
class PersonClass {
    var name: String;
    var birthYear: Int;

    init(name: String, birthYear: Int) {
        self.name = name;
        self.birthYear = birthYear;
    }

    func displayAge() {
        let age: Int = 2021 - self.birthYear;
        print("\(self.name) has \(age)");
    }
}

var personWithClass = PersonClass(name: "John", birthYear: 1972);
var personWithClassCopy = personWithClass;
personWithClass.birthYear = 2000;

// Will print `John has 21`
personWithClassCopy.displayAge()
// Will print `John has 21`
personWithClassCopy.displayAge()
```

Listing 8: Class Implementation and Original Mutation being Propagated

Despite the fact this a brand-new language, Apple has designed it to be backward compatible, which offers developers the smoothly transition from Objective-C. To cite Craig Federighi during WWDC 2014, "Swift is compatible with Cocoa and CocoaTouch, build with the same compiler, same ARC memory management model and same run-time, which means that Swift can fit right alongside Objective-C and C on the same application".

Even though Swift has been widely adopted by developers, Objective-C still receives support from Apple on iOS and macOS frameworks. In fact, Objective-C is still the foundation of Apple's Operating Systems, and some Swift functions rely on Objective-C to work, as for example: Timer function scheduledTimer allows developers to fire certain functions at a given time and these are expressed by a selector, which is, by extension, an Objective-C mechanism.

4.2 User Interfacing Technologies

The UI has a really important role on any Software, since it acts as communication bridge between the end-user and the Application itself. This layer allows the input and output of information as well as the user's interaction across components.

To implement a UI layer, Apple ships its SDK with two powerful yet distinct frameworks: Cocoa and SwiftUI.

Initially all macOS and iOS GUI were implemented by an imperative method, with Cocoa and Cocoa-Touch, respectively, through the Interface Builder.

Some years later after the introduction of Swift, Apple has unveiled a declarative framework of User Interfacing called SwiftUI.

4.2.1 Cocoa and CocoaTouch

4.2.1.1 Architecture

Cocoa is an Application environment for both macOS and iOS Operating Systems. This framework has been initially introduced by NeXT computers in 1989, which was acquired by Apple whom decided to continue its work by shipping the Framework, until today, on their Mac's Operating Systems, the OS X[3]. Later on, with the introduction of iOS SDK tools, Apple developed CocoaTouch based on Cocoa's foundations.

Under the hood, iOS platform has a five layered architecture with distinct, as depicted by Figure 15. To summarize how they work and their responsibilities, from the low to the highest level on their hierarchy:

- **Core OS**: is the lowest level in the hierarchy and contains the System Kernel, which is based on Unix;

[3]OS X is now known as macOS

- **Core Services**: allows string manipulation, network communication, and provides applications access to Hardware features such as GPS, compass, accelerometer, gyroscope. This is the layer where Software Developers interact with Programming Languages like Objective-C or Swift;

- **Media**: provides multimedia features, such as video and audio playback, and implements application's animations;

- **CocoaTouch**: is an abstraction to the UIKit which provides objects to allow GUI definition and general application behavior

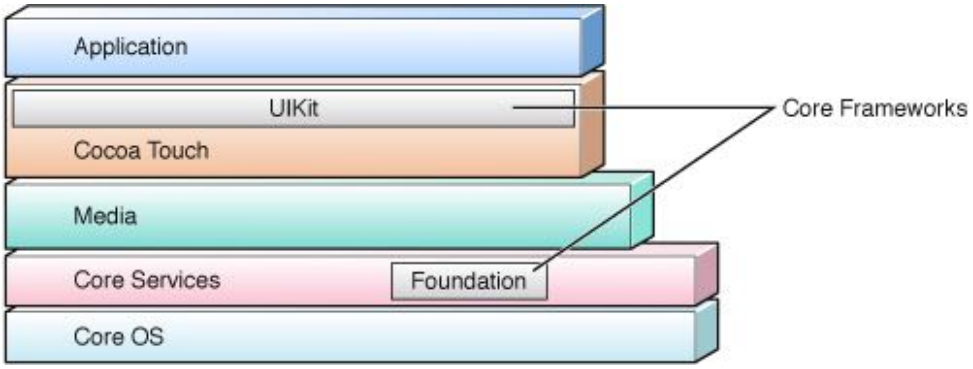


Figure 15: iOS Core Architecture [6]

4.2.1.2 Interface Builder

The User Interface implementation based on Cocoa and CocoaTouch is achieved by distinct methods: grammatically using code or with Interface Builder tool. However, in some cases, it is possible to build interfaces with both methods.

Interface Builder is exclusively available on Apple's own IDE, called Xcode, and allows an easy implementation of User Interfaces, based on Storyboards, by dragging and dropping components to their respective Views from the List of Objects, as depicted by Figure 16. This tool is highly recommended by Apple since it also helps to maintain a clean architecture, as for example MVC [25] - we will cover more information on this Architectural Pattern later during the Development Process 5.

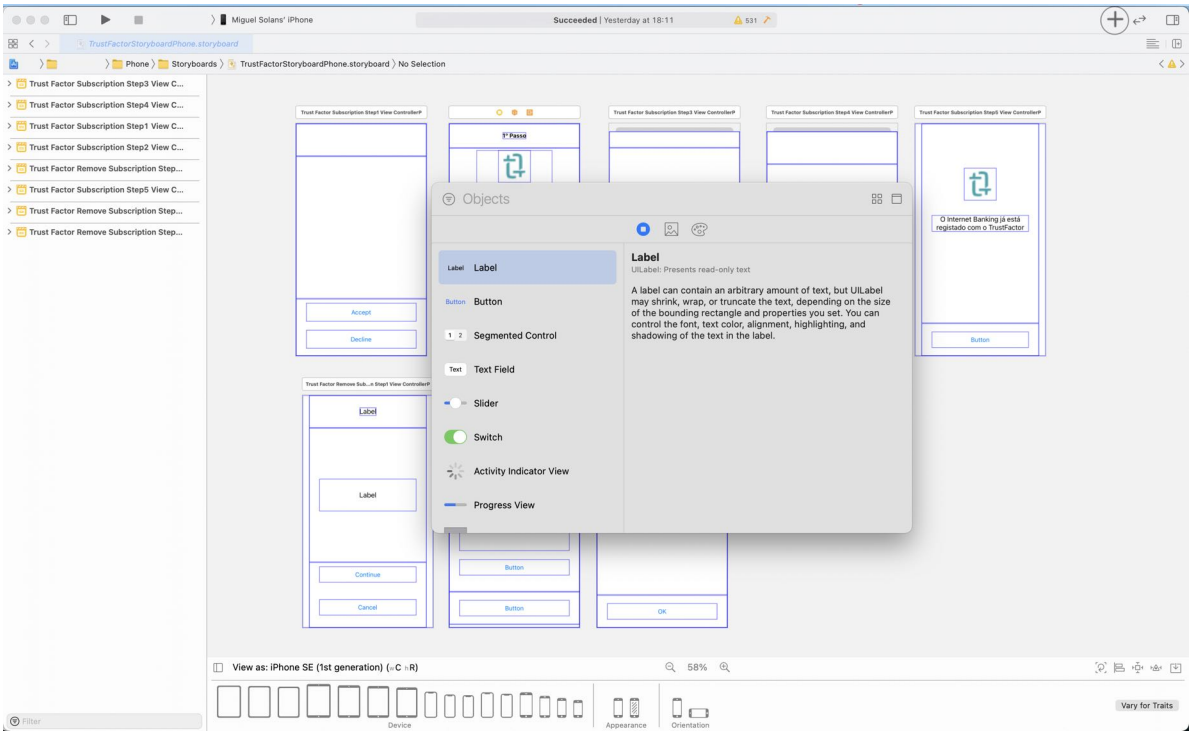


Figure 16: Adding UI Elements

In the perspective of UI and UX Design, a Storyboard has the main purpose of establishing a sequence between application states and define the overall flow between Views.

An iOS Application can be composed of several Views, also known as UIViews, that holds as many UI elements as needed to represent data. In order to allow users interact and navigate between Views, CocoaTouch also features *UIViewControllers* for this purpose. On Interface Builder, the flow between Views can be represented by Segues. The Figure 17, taken from Apple's official documentation, represents an hierarchy of elements made available by this framework.

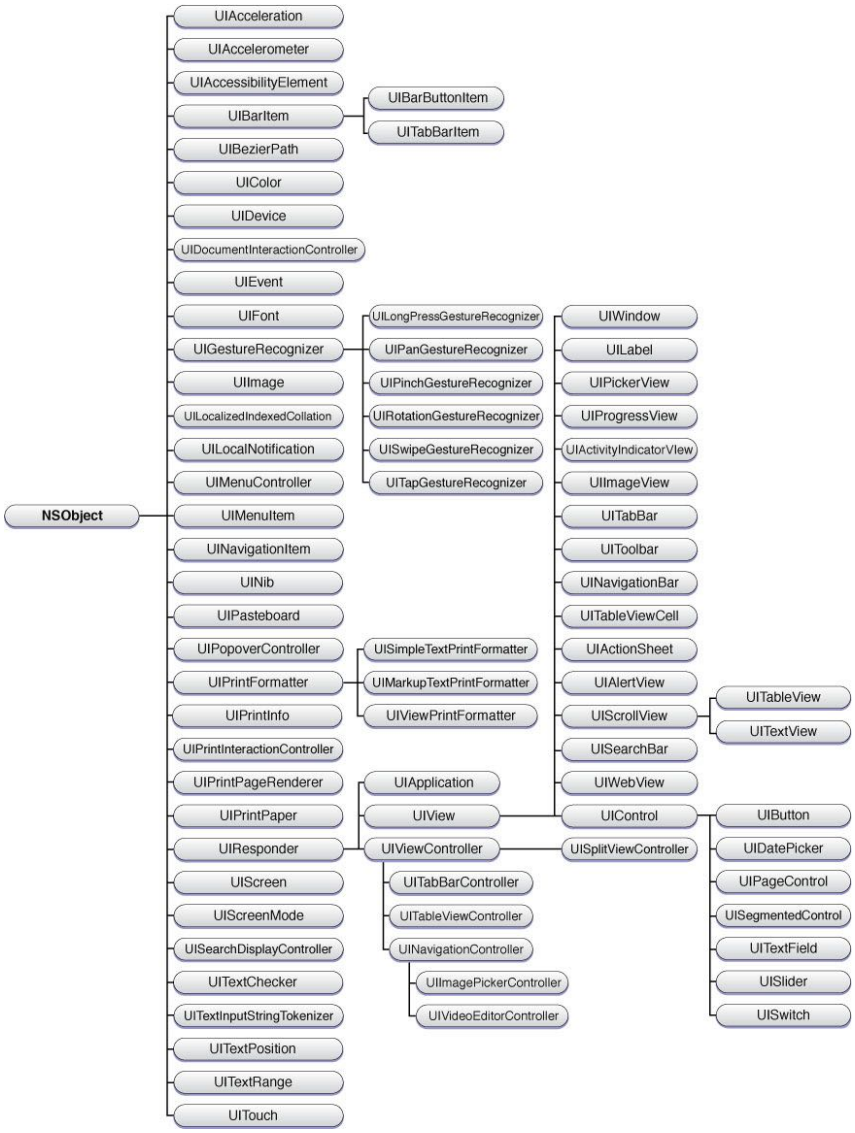


Figure 17: CocoaTouch Components [6]

4.2.1.3 Lifecycles

User Interfaces should have some dynamism and to achieve this in iOS Cocoa framework provides a mechanism to load and remove views from screen.

This mechanism is built on UIViewController Class and it is known as Lifecycles. The latter allows Software Developers perform additional configurations during different application states.

The Class UIViewController provides a set of methods invoked throughout the *Appearing*, *Appeared*, *Disappearing* and *Disappeared* App states. The available methods in each state are depicted by the Figure 18.

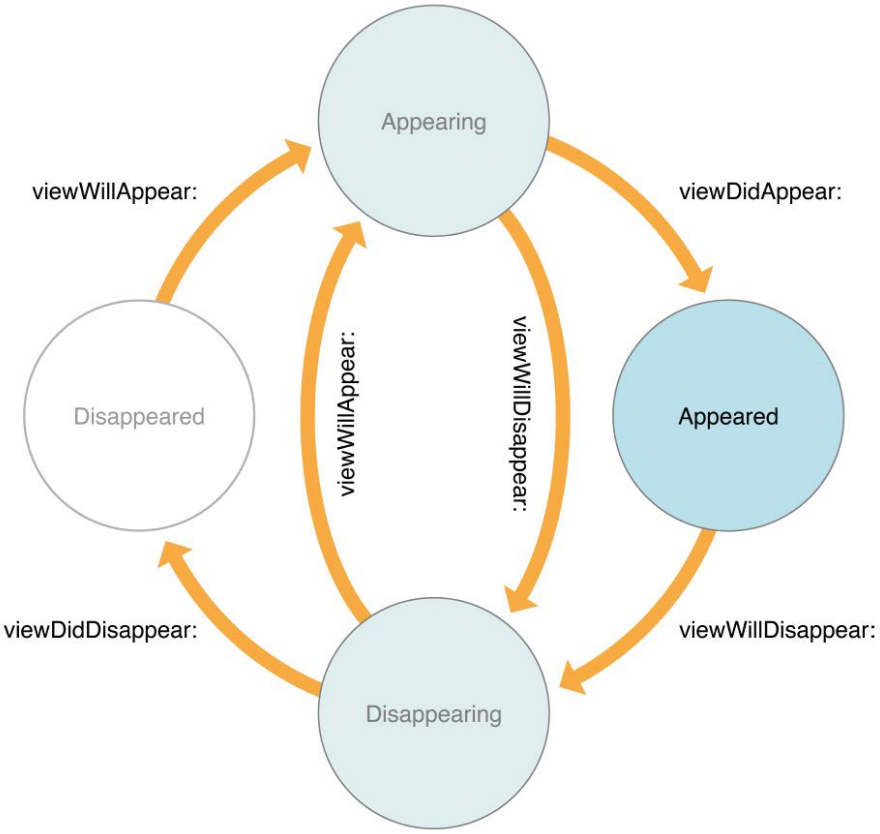


Figure 18: View Lifecycle [5]

In the beginning of a View lifecycle in Cocoa and CocoaTouch, a View is only loaded to the device memory by the View Controller when it's required [29] by the latter. As soon as the View-Controller has successfully loaded its view, the **viewDidLoad** is the first method to be invoked. This is one of the most important methods on a View-Controller and it is used to perform customizations or additional configurations [20] such as, for example, initializing variables and change or load Sub-Views. This method is only invoked once throughout the View-Controller lifecycle.

The **viewWillAppear**, as its name states, is invoked right before the View is loaded onto the View's hierarchy, or before it is presented on the device's screen, and before animations take place. This method is useful to perform certain operations before the view has been shown on screen, such as changing the screen orientation, apply different styles to the User Interface elements such as fonts or colors and in cases when the App relies on network communication it can be used to perform API calls.

At last, but not least important, the **viewWillDisappear** and **viewDidDisappear** can be used when the View is being discarded from screen. The first is executed before the View Controller has been removed from the View's hierarchy and it can be useful to store information automatically so it can be restored later on upon request, whilst the latter is more recommended to purge additionally resources required by the View-Controller.

4.2.2 SwiftUI

Back to 2014, when Apple introduced the new Programming Language called Swift, they had to make sure it could have access to UIKit, also known as CocoaTouch - the only available framework to develop Applications based on Graphic UI for the Apple ecosystem. However this has changed five years later when Apple unveiled, by the end of WWDC 2019 Keynote, a new Interfacing alternative to UIkit, the SwiftUI.

During this Keynote, according to Apple's executive, Crai Frederighi, SwiftUI has been developed entirely using Swift in order to take full advantage of their new Programming Languages features [7], which has introduced a whole new paradigm of Graphic UI to their ecosystem called Declarative Interfaces.

Compared to CocoaTouch, accordingly to the WWDC 2019's Keynote, this new Framework allows Application Developers to significantly reduce the amount of needed code to implement certain features and allows them to focus more on the App functionality while the Framework automatically deals with Animations, Accessibility settings, Dark Mode and Responsiveness - capability of adjusting to different screen sizes and orientation modes.

Considering SwiftUI is relatively new Framework means it is only supported by devices running iOS 13, macOS 10.15, WatchOS 6.0, or superior versions [24]. Given this fact, it is not feasible to use this Framework since the features we have to deliver, by Bank requirements, must support later versions of Apple's operating systems, starting from iOS 12 on-wards.

4.3 Frameworks and External Packages

Even though Apple restricts certain features from other Mobile Development Paradigms, under the Development of Native Applications it is possible to import third-party frameworks or libraries.

By using third-party libraries, developers are able to re-use generic and often well-matured code from other projects or developers to implement features more quickly and reduce development costs.

In iOS Application Development, there are two methods to import third-party code: by copying code or simply use a dependency manager that manages integration in projects and updates frameworks on demand.

In this chapter we will present CocoaPods, a famous dependency manager for Cocoa based Applications and AFNetworking library which we have used to implement networking communication to the Authentication and Transaction Validation services described during Development Process - section number 5.

4.3.1 CocoaPods

Like we have mentioned earlier in this chapter, CocoaPods is a third-party framework distribution and a dependency manager for both Swift and Objective-C Cocoa projects [12], meaning that it can be integrated not only in iOS but also macOS Applications.

CocoaPods isn't affiliated with Apple whatsoever, meaning that when we first install Xcode and Development Frameworks, if we want to integrate this dependency manager later we must install and configure it first. Before installing, according to CocoaPods documentation, the dependency manager is built with Ruby, so it is important to verify whether it is installed in our machine - in most cases it is, but if we use a Ruby Version Manager, the author recommends using the standard version of Ruby shipped with macOS.

Even after installation, when we generate a project with Xcode, the dependency manager will not be automatically integrated, meaning that we have to set it up by our own. Once CocoaPods is installed and created our project, we must Initialize it by running the "pod init"command in Terminal under the same directory of our project.

This command will generate a Podfile where we can define the libraries and frameworks we want to fetch from CocoaPods [4] to be later integrated in our project as Targets. The Listing 9 depicts an example of a Podfile where we are importing AFNetworking Pod with version 4.0.

```
target 'CocoaPodsTest' do
  use_frameworks!

  # Pods for CocoaPodsTest
  pod 'AFNetworking', '~> 4.0'

end
```

Listing 9: Example of a Podfile

Right now we have declared the dependency we want to integrate in our project, but we haven't yet installed it. To do such, under the same directory as our Podfile, we have to run the "pod install"command. The Pod utility will then download all dependencies from their respective repositories and create an Workspace file. We must always open our project using this file, as it contains targets for both our code and the integrated frameworks using Pods.

4.3.2 AFNetworking

Based on what we already know from CocoaPods sub-chapter, AFNetworking is a third-party Framework which can be installed by the dependency manager.

The AFNetworking is a networking library built for iOS, macOS, watchOS and tvOS devices. This library has been built with Apple's Foundation URL Loading System [1], which is the mechanism devices used to communicate with servers, identified by URLs, by using standard Internet protocols like HTTP.

This library works as an abstraction layer by extending Cocoa features and under the hood provides a modular architecture, as according to its author. The framework also has a wide community of developers, having reached 33 thousand stars on GitHub and it is maintained by nearly 320 contributors on the same platform.

[4] We can find Pods on CocoaPods' Website: https://cocoapods.org

Due to the fact this library is built upon Cocoa technologies and provides an abstraction layer with modular architecture to communicate with servers, it requires less lines code than URLSession for the purpose, as depicted by comparison code in Listing 10 of a simple GET HTTP request obtained from GitHub Gist[5].

```
// AFNetworking
[[AFHTTPSessionManager manager] GET:@"http://httpbin.org/ip" parameters:nil
success:^(NSURLSessionDataTask *task, id JSON) {
        NSLog(@"IP Address: %@", JSON[@"origin"]);
} failure:^(NSURLSessionDataTask *task, NSError *error) {
        NSLog(@"Error: %@", error);
}];


// NSURLSession
NSURL *URL = [NSURL URLWithString:@"http://httpbin.org/ip"];
NSURLRequest *request = [NSURLRequest requestWithURL:URL];
[[NSURLSession sharedSession] dataTaskWithRequest:request
completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
        if (error) {
                NSLog(@"Error: %@", error);
        } else if (data && [data length] > 0) {
                NSError *JSONError = nil;

                id JSON = [NSJSONSerialization JSONObjectWithData:data
                        options:0 error:&JSONError];

                if (JSONError) {
                        NSLog(@"Error: %@", error);
                } else {
                        NSLog(@"IP Address: %@", JSON[@"origin"]);
                }
        }
}];
```

Listing 10: Simple GET Request with AFNetworking and NSURLSession

4.4 Summary

This section has been focused on the technologies Apple provides in order to build Native Applications.

We have introduced basic concepts of their Programming Languages, Objective-C and Swift, also as the differences between each other.

Secondly, Cocoa has been introduced as framework to develop User Interfaces for both iOS and macOS, as also SwiftUI - the newest Interfacing framework. We have discarded the latter as it isn't matured as Cocoa and due to the requirement of our project which must support a wider range or iOS versions.

[5]AFNetworking vs NSURLSession in Objective-C: https://gist.github.com/AlamofireSoftwareFoundation/bb16a491b2709a8476e2

Despite the fact that this chapter has been more focused on official technologies, we have also introduced CocoaPods manager which provides access to third-party Frameworks. We have also hereby introduced AFNetworking library, which we have used to perform networking requests - download and upload data to a REST Server.

5 Development Process

The Development Process chapter documents the studied concepts about iOS Native Development whilst discussing the TrustFactor Authentication mechanism integration in the Home-Bank application.

In this chapter we will also cover topics related to good programming practices such as Design Patterns and Architectural Patterns and the differences between each other and how they fit into the Home-Bank application.

5.1 Architecture and Design Patterns

Software Development grew over the years and devices have become more powerful to keep up with the market demands and to perform highly complex computational steps. Due to this fact, Software has gained a certain level of complexity over time and therefore the development phase has become more difficult and time consuming.

To address these issues, Design Patterns and Architectural Patterns have emerged, providing common solutions to recurring problems. In this section, we will cover the most important Design Patterns and Architectural Design Patterns used during the Development Process of this project.

5.1.1 Design Patterns

A Design Pattern provides a scheme for refining sub-systems or relationships between them and the ability to address general design problem under a given context [32]. By other words, Design Patterns articulates how various components or classes collaborates with each other in order to fulfill a desired functionality. For example, Cocoa and CocoaTouch makes effective usage of Design Patterns at its core, by providing a variety of abstract classes and functions used to solve recurring problems in a particular context [21].

There are many Design Patterns that can be applied to iOS Software Development, either Objective-C or Swift, and the most commonly used patterns in our work will be the Adapter, Observers and Singletons.

Cocoa and CocoaTouch, as we have previously stated, provides a set of functions whose behaviors can be overridden, however some classes interfaces might not be compatible with one another. To cope with

this issue, the Adapter Design Pattern allows classes with incompatible interfaces to work together. A common implementation of this Pattern can be found in CocoaTouch Protocols, such as *UISearchBarDelegate* which provides a set of methods to cope with user input.

Observer Patterns define a one-to-many relationship dependency between objects, so that when one specific object state changes the others are notified [21]. The Observer Design Pattern is also known as Publish-Subscriber, where Subscribers listens for changes on the Publisher. This Design Pattern is implemented by a commonly used Mechanism, *NSNotificationCenter*, which can be used, for example, to execute certain methods when the Application enters or leaves foreground.

Singleton Design Pattern is the easiest one to gasp. This Design Pattern ensures that one class has a single instance of a certain object. One application example of this Design Pattern in AppKit framework is NSApplication wrapper which has the main purpose of managing the main event loop and resources used by the application objects: application window's and menus, dispatching application state events [8], among other tasks.

5.1.2 Architectural Design

In contrast to Design Patterns, Architectural Designs express a fundamental structural system organization. That being said, Architectural Designs provide a set of predefined sub-systems, delegate their responsibilities and define rules or guidelines for organizing relationships between components [32]. Because of the fact the system is split into different components, Architectural Designs also promote components re-usage under different scopes, without having the need to repeat code over and over again.

We have previously stated that when building iOS Applications upon CocoaTouch UIKit technology, Apple encourages the usage of MVC Architectural Design, however we have not yet given a clear explanation on how it works and its entities.

The MVC Architecture is composed by three components with different purposes, and its acronym stands for Model-View-Controller. The Figure 19, taken from Apple's official documentation, depicts an overview on how these components interact with each other.

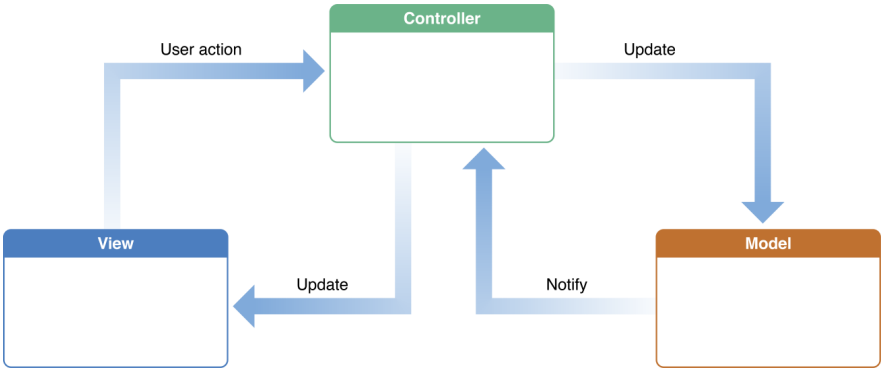


Figure 19: Model-View-Controller Architecture [4]

The **Model** component encapsulates data and usually defines business logic on how it should be manipulated and processed. In other practical words, in our application, the Model represents a definition of JSON data we are expecting to retrieve from an API, serializes it to a class for later access or on-screen representation. However, as we can observe from Figure 19, the View layer should not access the Model layer directly, the communication between these components should be mainly done via a Controller entity, which requests the data access or change and therefore notifies the View.

Like its name suggests and as we have previously stated during the Cocoa and CocoaTouch chapter, a **View** layer represents UI components such as images, switches, labels, inputs and other available components. The View can also represent data fetched from an external networking API and receives user inputs to be later sent over to their respective Controller layer.

As we have seen, the **Controller** acts as a intermediary layer between Views and Models. All our logic is implemented on a Controller, that should respond to User Inputs sent from the View, which based on those inputs should fetch or change the Model representation. Controllers can also perform additional setups and manage the overall Application.

5.2 TrustFactor Integration

As we previously discussed on the State of Art, TrustFactor is a stack of services who cooperate between one another, by acting as a middleware layer on the home-banking services, to provide a Strong Authentication and Validation mechanism for fund transactions.

These services are mainly implemented on the Backend-side however, to provide access to this mechanism, an integration on user's apps is needed so the business logic can function as expected.

Throughout this section, we will discuss the On-boarding Process and the logic behind the Transaction processing.

5.2.1 Backoffice

The TrustFactor stack provides a Backoffice simple to use and to manage user subscriptions to the Strong Authentication mechanism. This Backoffice also allows Bank Institutions to manage parameters to be used during the Risk Analysis of the transaction execution.

As soon as the TrustFactor is integrated as a middleware Authentication layer, it is possible to use the Web Application to define which Transactions should rely on TrustFactor authentication service. In our case scenario, by Bank Institution requirement, this Strong Authentication mechanism should only be applied to Same Bank Transfers, also known as National Transfers. For this to happen, via Backoffice, we must set an active action for the Transaction ID "TransferenciasNacionais", as depicted by Figure 20.

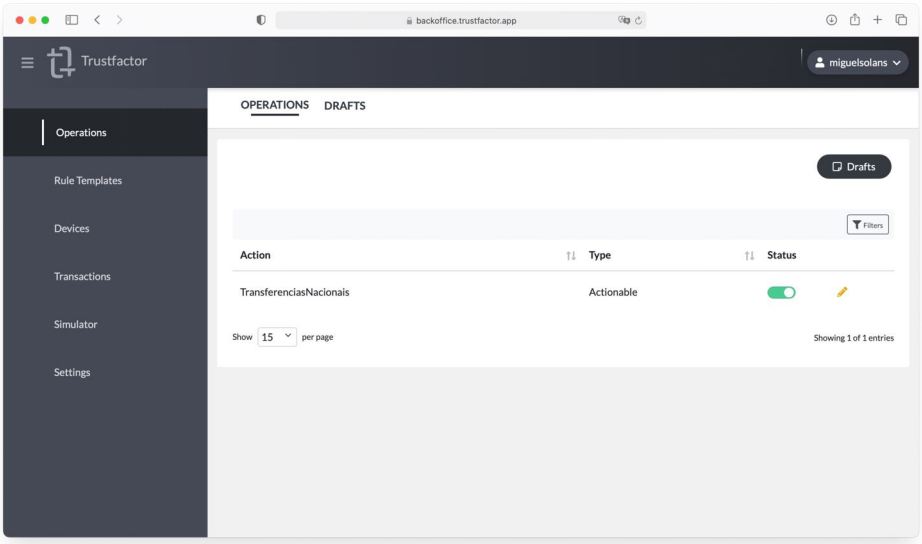


Figure 20: Transaction Identifiers Depending on TrustFactor

Following to that, the Backoffice displays a set of parameters that can be activated for Risk analysis. In order to validate the integration with Home-Bank with TrustFactor services, we will use the parameter based on the transaction amount.

As depicted by Figure 21, filters based on the transaction amount are set by adding limits to the labels Low, Medium and High Risk of execution. For testing and future examples throughout the Development Process we have defined that values ranging from 0 to 49 should be classified as Low Risk, 50 to 99 as Medium Risk and transactions superior to 100 should be considered as a High Risk transaction.

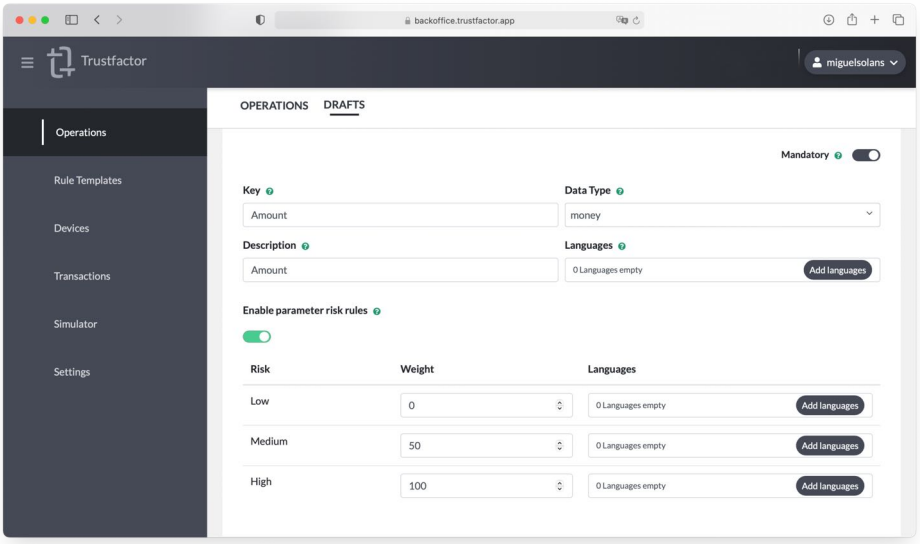


Figure 21: Risk Limits Definitions

5.2.2 REST API

For integrating the TrustFactor mechanism, this service provides an REST API composed by seven endpoints described by the Tables which have been added to the attachments section B.

Each endpoint has a different responsibility within the service, and each can be understood by the following list:

- **Requirement**: returns a mapping of values to asses the user subscription to TrustFactor authentication;
- **Register Code**: requests a TrustFactor registration codes or QRCode to be used later on;
- **Refresh Register Code**: fetches for a new registration codes when the previous request has been expired;
- **Check Register Code**: returns a validation mapping code for token validation - whether it has expired, used of invalid.

5.3 TrustFactor Subscription

Unlike other existing authentication mechanisms (SMS and Positions), by the bank's request, the end-user must be the one to decide whether the Homebanking platform should use TrustFactor as a Strong Authentication mechanism or not.

However, migrating existing solutions to more recent ones can be a complex process that should be taken into maximum consideration, otherwise it can lead to short or long-term Software malfunction. To tackle this problem, we have implemented an *Onboarding* process within the client's application.

In Software, an *Onboarding* process is defined as a flow of screens when a user first accesses a newly installed Software or it can also be used to help users to configure certain features for the very first time.

In our case, to develop the TrustFactor onboarding process, we had to take into account the following requirements:

- **Terms and Conditions**: prior to subscription, the user should be aware and accept the terms and conditions of TrustFactor service;
- **Installation**: in order to successfully subscribe the TrustFactor authentication service, the user should install the TrustFactor agent on his/her device;
- **Validation**: the subscription service should only be done after performing a successful validation with one of the existing authentication mechanisms - SMS or Positions;
- **Association**: the Homebanking app and the TrustFactor agent are standalone applications, therefore the App must feature mechanism to associate one with the other.

To tackle the requirements we have mentioned above, we have designed five different *Views* within the app, where the last allows us to inform the user on the success of the service subscription, resulting on a flow of wire-frames, starting from the left to the right, as depicted by the Figure 22.

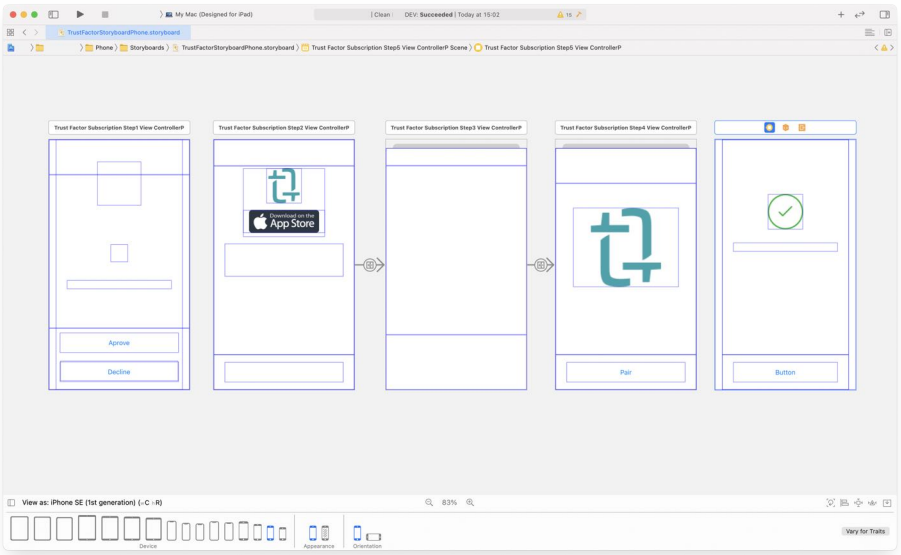


Figure 22: Onboarding Storyboard

5.3.1 Terms and Conditions

During the subscription flow, the first View has the main purpose of presenting the "Terms and Conditions"of the Strong Authentication Service to the users and receive their acceptance.

Since the TrustFactor is associated on-device, after it takes place, the onboarding process should not available anymore, and preceding with it will result on a service error. To address this, inside the View-Controller, we check whether there is an active subscription or not, by making an HTTP request to the API's endpoint Requirement.

As we can observe from the indexed table in Appendice's Section C, the Requirement endpoint responds with a Number, expressed as an Integer, which indicates the subscription status of the account. The requirement can assume different values, which are calculated on the Backend-side using a binary sum of the values mapped to an Enumerator as depicted by the Listing 11.

```
typedef enum Requirement {
    NotRequired = 1,
    RequiresSubscription = 2,
    Subscribed = 4,
    NotRequiredForTransaction = 8,
    RequiredForTransaction = 16,
    UnknownRequirement = NSUIntegerMax
}Requirement;
```

Listing 11: Requirement Value Mapping

Considering we want to verify whether the use has subscribed the service or not and the Requirement field represents a binary sum of numbers ranging from 1 to 16 (in some cases it can assume the value 20, if the account has Subscribed to the service and the transaction does not require TrustFactor), we perform an And Bit-Wise operation between the Subscribed mapped value and the result from the data API. If this condition is said to be true, we redirect the user to the screen depicted by Figure 23.

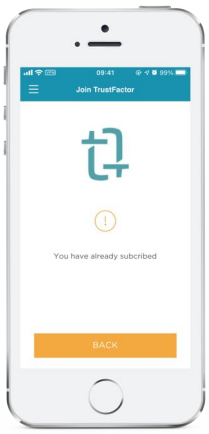


Figure 23: Subscription already active

5.3.2 Installation

In case the account does not have an active subscription, after accepting the "Terms and Condition"of the Strong Authentication service, to associate the user's contract to the device, the installation of the TrustFactor Agent Application is mandatory. As such, we added another step requesting the user to install the App with an App Store image acting as a button for a quick access to the TrustFactor Agent installation page for installation, as depicted by the Figure 24.

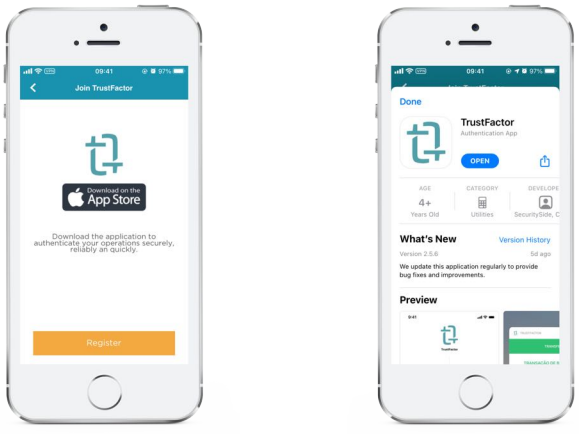


Figure 24: TrustFactor Agent Installation Step

To support this a direct interaction between the Home-Bank App and the App Store, we have used a core API provided by Apple on the UIKit SDK called StoreKit.

In order to use StoreKit in our View-Controller, the latter must inherit from the Delegate SKStoreProductViewControllerDelegate. With this delegate, one can load an App Store View through a Push Modal, by referring the Application ID we want to display. For example, to load TrustFactor, by its own identifier 1525877833, we have used the following code excerpt depicted by Listing 12

```
SKStoreProductViewController* spvc = [[SKStoreProductViewController alloc] init];
[spvc loadProductWithParameters: @{
    SKStoreProductParameterITunesItemIdentifier : @1525877833
} completionBlock:nil];
```

Listing 12: Load App Store View from an Action

5.3.3 Validation

The Subscription and Association process to the TrustFactor Agent application can only be done after validating the user integrity. We can try to assure this by using an existing security mechanism.

In order to validate the user's integrity we send an SMS to the number associated with the Bank account with a Token on its body every time the user's jump right onto this step. Upon receiving the Token via text-message, the user is prompted to type it into the text-field, as depicted by Figure 25.

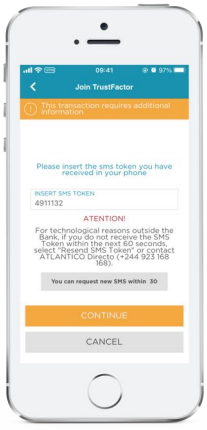


Figure 25: SMS Token Credential Step

After inserting the Token into the text-field and pushing the "Continue"button, the Token is sent over over for further validation via an HTTP POST request to the Bank's REST service. If the credential is said to be valid, or in accordance with the generated Token by the Backend-side, we request for a valid TrustFactor registration code, to be used on the next step, sent by segue mechanism.

5.3.4 Association

The Association step is considered to be the last on Strong Authentication service subscription. It's in this particular step the Home-Bank user contract is associated with TrustFactor Agent.

The iOS architecture has been built upon a very strict security model where Apps are isolated to their own sandbox, which means they can only access their own data and not personal data nor other apps'. This comes with a little of a drawback to the TrustFactor implementation because the Home-Bank app can't access the Agent directly.

To cope with this problem, the Register-Code endpoint provides us an URL that acts as a deep-linking between apps. A deep-link is a common URL pointing towards a specific content or, in our case, an App installed on the device. Like browser's URL's, deep-links also supports query strings, which has helped us to pass data from the Home-Bank app to the TrustFactor Agent, as a payload query parameter. The Listing 13 depicts the deep-linking URL structure to the TrustFactor Application Agent.

```
open.trustfactor.app/register? payload= AssociationToken
```

Listing 13: Deeplink URL Structure

In order to finally pair the Bank contract with the TrustFactor Agent the user should push "Pair"button. This button has a deep-link URL associated with itself, that once triggered tries to open it by with help of NSURL class. Once the URL has been successfully open, the Application TrustFactor should open and prompt.

However, to ensure security, TrustFactor registration code has a time-to-live timer, meaning that after a certain time of its creation the token is rejected by the Backend services. This timer is expressed in seconds and is sent over the response JSON body of Register-Code endpoint.

Given the fact these Tokens can expire before the User has successfully paired their contract with TrustFactor Agent, the Backend services provides an endpoint that given the expired Operation ID, returns a new Token for a successful association. We then discard the expired code and use the newest for association.

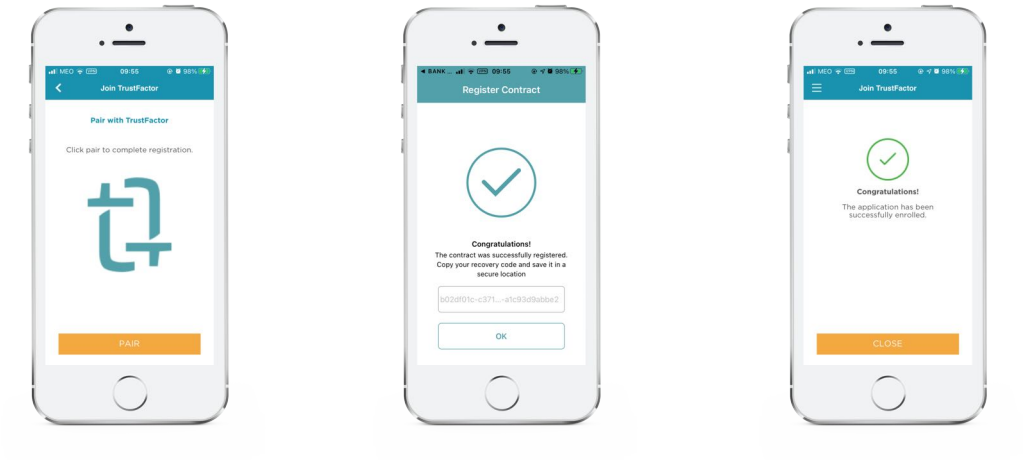


Figure 26: TrustFactor Agent subscription flow

The Figure 26 depicts the association flow, where the outer left and right screens represents the Home-Bank Application, whilst the middle one the TrustFactor Agent.

Although the TrustFactor Agent presents a successful screen, something wrong can happen throughout the subscription. To make sure the User is aware of the status, we have implemented one last screen on the on-boarding flow. This screen is shown to the user based on the result of an HTTP POST request to the Check Register Code endpoint, where the Unique ID parameter refers to the Operation ID we obtained from Register Code. The API based on this ID then responds with a Status number and a possible message error.

The Check Register Code status can assume values ranging from 1 to 4, which represents the following cases:

1. The Backend services required client to once again validate the subscription by re-sending the Operation ID;

2. The subscription was successfully completed;

3. and 4 indicates an error expressed in ErrorMessage property from the JSON response body

Given the fact subscription process only takes place when the user is redirected from the Home-Bank App to the TrustFactor, the first leaves foreground until the User returns back and, if he does, the App performs the Network call and the validation we have mentioned above. This is done by registering an Observer on the *UIApplicationDidBecomeActiveNotification*, by recurring to the *NSNotificationCenter* mechanism, depicted by the Listing 14, where the **becomeActive:** is a selector for a function block with the validation logic.

```
[[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(becomeActive:)
                        name:UIApplicationDidBecomeActiveNotification object:nil];
```

Listing 14: Load App Store View from an Action

If the TrustFactor subscription is then said to be successful, the Observer for *UIApplicationDidBecomeActive* notification is then removed and the App presents the last screen depicted by Figure 26 to the user.

5.4 Transaction Process

As we have previously stated, the Home-Bank Application has been built on a MVC Architecture, which means business model, UI implementation and logic are separated entities. Besides this, the MVC Architecture allows us to implement and re-use components under different environments without the need of re-writing behaviors multiple times.

With this in mind, the Home-Bank Application inherits from four step Generic Controllers to cope with the transaction process, regardless of its type. The Generic Transaction Architecture is depicted by Figure 27 and bellow we introduce each step responsibility:

- Step 1: **Input Fields** for the transaction details, such as currency, amount and destination. After editing these fields, a validation of the input information is performed before continue;

- Step 2: **Transaction Overview** displays a summary based on the input information from previous step and requests the authentication method for transaction;

- Step 3: **Transaction Validation** prompts user for transaction approval with multi-factor authentication;

- Step 4: Presents a **Status Screen** to the user, whether the transaction has been successfully executed or not.

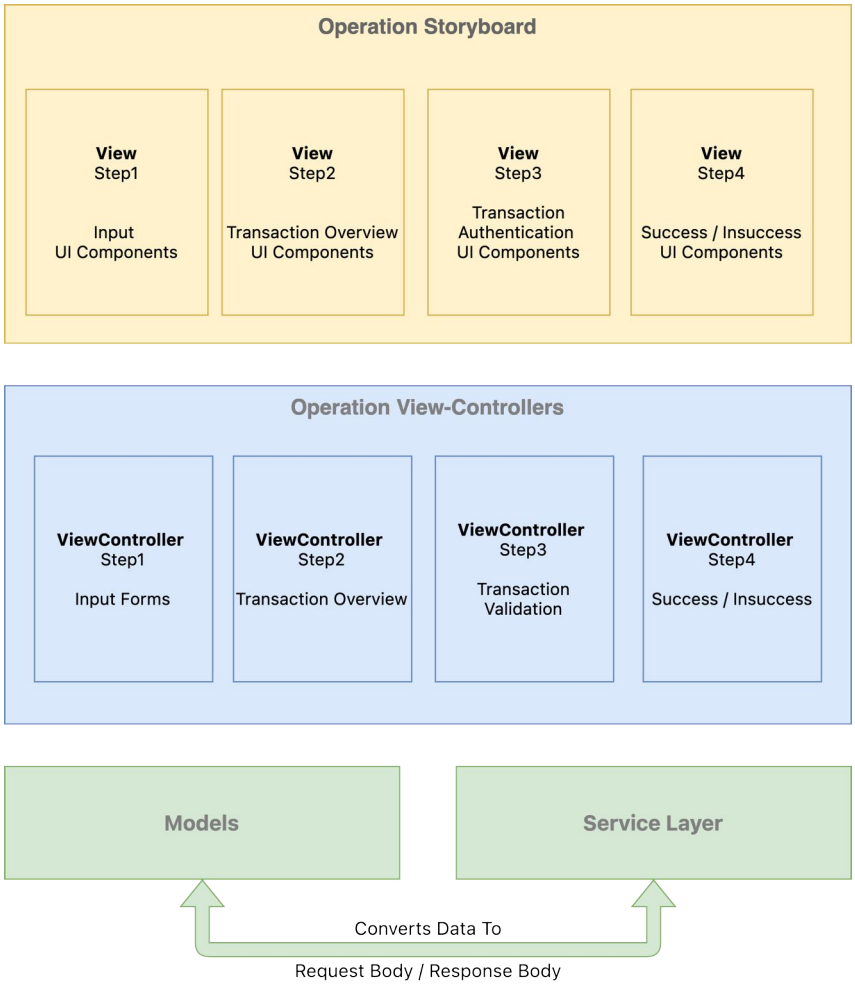


Figure 27: Architecture of Fund Transaction Flow

Given this, the integration of the Authentication mechanism via TrustFactor was mainly performed at the Step 2 and the execution process on the Step 3.

As we have previously discussed, the subscription to this new mechanism it's not mandatory and the mechanism should only be triggered if the transaction requires it and the user has an active TrustFactor subscription.

In contrast to other existing authentication mechanisms, SMS Token and Positions, the authorization via TrustFactor does not require a direct input from the user in the App. As such, we have implemented a stand-alone XIB that presents to the user three different informative images under different circumstances, which we will discuss later.

From the left to the right, the Figure 28 depicts the transaction flow in accordance to the architecture described by Figure 27, where in the first step the user must type the details for the fund transaction - BIN, amount, description and so on. The next screen presents an overview of the transaction details. Since this second step is used across other type of transactions, we dispatch an HTTP request to the Requirement endpoint and perform, once again, the And Bit-Wise operation to check the subscription

status and TrustFactor Requirement for the transaction. If both conditions are said to be true, we then load the TrustFactor transaction screen.

Once the transaction authorization screen is loaded, the third step, the App dispatches an HTTP POST request to the TrustFactor Create Transaction endpoint, with the transaction data and the type of transaction and operation ID's. The endpoint will then respond with a String ID in the data body property, to be used later on to verify the transaction state, and a Push Notification will be sent to the device which has been previously associated with the contract, informing that a transaction request has been made, as depicted by Figure 28.

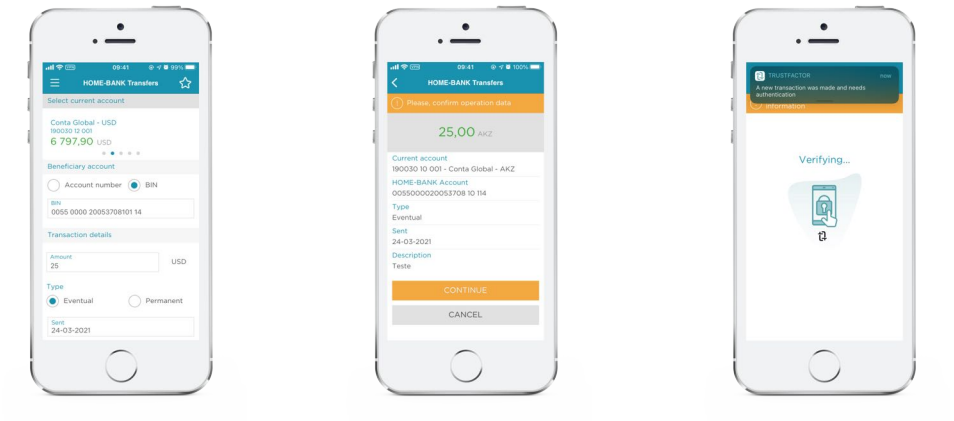


Figure 28: UI Fund Transaction Flow

As soon as the device receives a Push Notification, if the user clicks on it, he or she can have an overview on the associated risk of the transaction execution, based on the defined parameters. User will also be prompted to reject or accept the transaction execution. To exemplify, the Figure 29 depicts transaction requests with the amount of 25, 150 and 500 AKZ, being labeled as Low, Medium and High Risk Transactions, respectively.

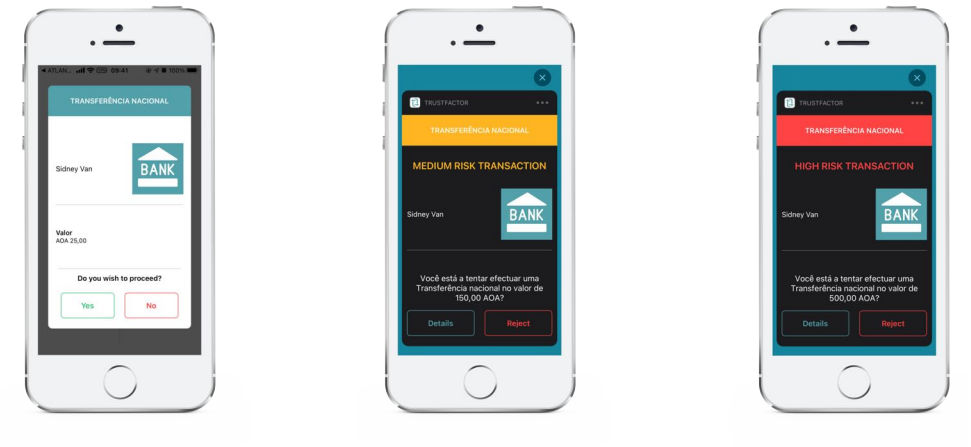


Figure 29: Example of Transaction Classifications

Because of the fact that iOS restricts each App to its own sand-box, as we have previously stated, the TrustFactor Agent is not able to interact directly with the Home-Bank App. Given this fact, after the Push Notification has been sent, the only possible method to verify whether the transaction has been accepted is by performing a long-polling HTTP request to the Check Create Transaction endpoint by passing the ID given by Create Transaction over the request body. The Check Create Transaction will then validate the execution status on the Backend and return values ranging from 1 to 5. We display different TrustFactor images, as depicted by Figure 30, based on these values which represent the following states:

1. **Pending**: The transaction has not been confirmed yet;

2. **Accepted**: The transaction has been accepted by the TrustFactor Agent;

3. **Expired**: The transaction has exceeded maximum time and therefore has been cancelled;

4. **Error**: The transaction has been cancelled due to an error;

5. **Declined**: The transaction has been declined by the TrustFactor Agent

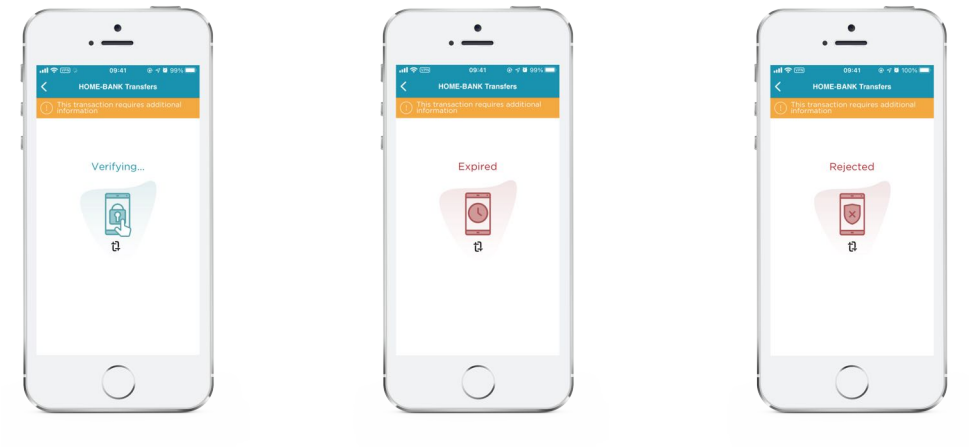


Figure 30: TrustFactor Execution Status

5.5 Summary

The Development Process has been focused on the problems and their solutions, whilst displaying examples of the end-result.

It started by exploring common Design Patterns used under the scope of iOS Development, such as Adapter, Observer and Singletons, followed by the Architectural Pattern called Model-View-Controller.

After introducing these development practices, we began discussing the integration of Strong Authentication mechanism via TrustFactor in the Home-Bank Application.

6

Conclusions and Future Work

As an outcome to this Dissertation, the Chapter Conclusions and Future work will initially summarize the research and discuss the project milestones, as well as a general appreciation of the executed work.

To have closure, and considering this project has been accepted by the client to enter Production phase, we will establish future work to be made in order to improve the App's security, maintainability and enhanced User Experience.

6.1 Overall Conclusions

As we have previously stated, this Dissertation had the main purpose of integrating a Authentication layer on a existing Home Banking Application, by recurring to TrustFactor service stack, and investigate the development paradigms of iOS Applications.

Even though it was a requirement to work with Native Technologies, we started by researching other existing frameworks that can be used to build iOS Applications. On Mobile Development Paradigms, we have found three different approaches: Progressive Web Applications Development where we can use Web based technologies to create dynamic Applications that run on Web Browsers; Cross-Platform Application Development where the same code-base can be shareable across multiple Operating System environments, having a nearly Native Application experience and reduced costs of development. We ended up with Native Application Development due to the fact the App was already made and also because of limitations imposed by the other paradigms - like absence of Push Notifications, reduced User Experience metrics, Application sizes, slow access to new features and worse integration with the overall Operating System functions.

With the purpose of understanding the project Architecture and how it works internally, we have introduced Apple's Objective-C and Swift programming languages focusing on their syntax differences, followed by the CocoaTouch *framework* which allows one to implement Application's Graphical User Interfaces - or also known as GUI. Besides this framework, even though we have not used it due to its API requirements (target limited to 13 for iOS on-wards), we have also introduced SwiftUI as an Apple's official alternative.

Programming Languages and other Technologies are often an abstraction layer we can customize in order to shape our business logic into one Application. Therefore, sometimes these require more code to execute simple tasks, hence developers may add external dependencies to speed up development. For this matter, we have introduced one commonly used Package Manager, the CocoaPods, and a HTTP client named AFNetworking which reduces complexity compared to the built in foundation's NSURLSession.

After having done an extensive research work on technologies and important concepts used by Home-Bank App, we were able to have a better insight into iOS Native Development and successfully integrate TrustFactor stack on the Client side.

6.2 Future Work

As we previously mentioned, the Home-Bank Application is an existing product and the TrustFactor security layer we hereby presented has been accepted by the Client to enter App Store distribution. Thus, it is important to present hereby certain aspects that should be improved as time goes on.

First off and considering the wide adoption of Swift in comparison to Objective-C, we would like to start migrating our code base, starting from generic functions, in order to stay up-to-date with new market standards. This would not only improve App performance as would also ease of maintainability since Swift syntax is easier to understand for both experience and new developers.

When it comes to functionalities, TrustFactor's subscription services allow developers to present a QR code to be read in order to pair with other devices, rather on-device. This could be easily done by parsing a 64 base format string to an UIImageView output. By implementing this feature, Users could opt to choose a different device to associate their contract with, and have two devices for different use cases - one to request transactions and another to authorize them.

Right now, every transaction made with TrustFactor has a timer associated with it. If an User takes more time than what he his supposed to, he will be presented an Expiration Error screen and would be required to fill-in the transaction form once again. This can cause an impact on User Experience metrics that could be easily addressed by allowing users to request for a new TrustFactor token with the same data at least once - which would make a great new feature in the future.

Bibliography

[1] AFNetworking. *AFNetworking*. url: https://github.com/AFNetworking/AFNetworking (cit. on p. 35).

[2] Altexsoft. *The Good and The Bad of Xamarin Mobile Development*. 2020. url: https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native/ (cit. on p. 20).

[3] J. Andress. *The Basics of Information Security - Understanding the Fundamentals of InfoSec in Theory and Practice*. Ed. by Elsevier. Elsevier, 2011. isbn: 978-1-59749-653-7 (cit. on p. 9).

[4] Apple Inc. *Model-View-Controller*. url: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (cit. on p. 39).

[5] Apple Inc. *UIViewController | Apple Developer Documentation*. url: https://developer.apple.com/documentation/uikit/uiviewcontroller (cit. on p. 33).

[6] Apple Inc. *What Is Cocoa?* url: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html (cit. on pp. 30, 32).

[7] Apple Inc. *WWDC 2019 Keynote*. 2019. url: https://developer.apple.com/videos/play/wwdc2019/101/ (cit. on p. 34).

[8] Apple Inc. *NSApplication*. url: https://developer.apple.com/documentation/appkit/nsapplication (cit. on p. 39).

[9] S. Bosworth, M. Kabay, and E. Whyne. *Computer Security Handbook*. Sixth Edit. John Wiley Sons, Inc, 2014. isbn: 978-1-118-13410-8 (cit. on p. 10).

[10] D. Britch. *Xamarin.Forms Supported Platfroms*. 2020. url: https://docs.microsoft.com/en-us/xamarin/get-started/supported-platforms (cit. on p. 20).

[11] N. Chrzanowska. *React Native Pros and Cons - Facebook's Framework in 2021 (Updated)*. 2019. url: https://www.netguru.com/blog/react-native-pros-and-cons (cit. on p. 18).

[12] CocoaPods. *CocoaPods - Getting Started*. url: https://guides.cocoapods.org/using/getting-started.html (cit. on p. 34).

[13] Cordova. *Architectural overview of Cordova Platform*. url: https://cordova.apache.org/docs/en/10.x/guide/overview/ (cit. on p. 19).

[14] Cordova. *Architecture Overview*. 2020. url: https://cordova.apache.org/docs/en/latest/guide/overview/index.html (cit. on pp. 17, 19).

[15] Flutter. *Flutter architectural overview*. 2020. url: https://flutter.dev/docs/resources/architectural-overview (cit. on pp. 21, 22).

[16] R. Gravelle. *Pros and Cons-Platform Mobile Development Frameworks*. 2015. url: https://www.htmlgoodies.com/mobile/pros-and-cons-of-cross-platform-mobile-development-frameworks/ (cit. on p. 19).

[17] C. Griffith. *What is Apache Cordova?* url: https://ionic.io/resources/articles/what-is-apache-cordova (cit. on p. 18).

[18] S. Gutherie. *Microsoft to acquire Xamarin and empower more developers to build apps on any device*. 2016. url: https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device (cit. on p. 20).

[19] Y. Horbenko. *Mobile Development: Choosing Between Native, Web, and Cross-Platform Applications*. url: https://steelkiwi.com/blog/how-choose-correct-platform-mobile-app-development (cit. on p. 16).

[20] A. Inc. "Work with View Controllers". In: (2018). url: https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/WorkWithViewControllers.html (cit. on p. 33).

[21] A. Inc. *Cocoa Design Patterns*. url: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html (cit. on pp. 38, 39).

[22] A. Inc. *Programming with Objective-C*. 2014. url: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html (cit. on p. 24).

[23] A. Inc. *Swift*. url: https://swift.org (cit. on p. 27).

[24] A. Inc. *SwiftUI - Declare the User Interface and Behavior for your app on every platform*. 2020. url: https://developer.apple.com/documentation/swiftui (cit. on p. 34).

[25] A. Inc. *Xcode - Interface Builder Built-In*. url: https://developer.apple.com/xcode/interface-builder/ (cit. on p. 30).

[26] J. Johnson, D. Britch, and C. Dunn. *What is Xamarin?* 2020. url: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin (cit. on p. 20).

[27] A. Manchanda. *Where Do Cross-Platform App Frameworks Stand in 2021?* 2020. url: https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/ (cit. on p. 17).

[28] S. Martin. *How Much Does It Cost To Build a Cross-Platform Application in 2021*. 2021. url: https://medium.com/flutter-community/how-much-does-it-cost-to-build-a-cross-platform-application-in-2020-6f07c941d666 (cit. on p. 16).

[29] M. Neuburg. *Programming iOS 12 - Dive Deep Into Views, View Controllers, And Frameworks*. Ed. by R. Roumeliotis. 9th ed. October. O'Reilly Media, Inc., 2018. isbn: 9781492044635 (cit. on p. 33).

[30] OneLoin. *What is Multi-Factor Authentication (MFA)?* 2021. url: https://www.onelogin.com/learn/what-is-mfa (cit. on p. 6).

[31] D. Ortinau, D. Britch, and C. Dunn. *Limitations of Xamarin.iOS*. 2020. url: https://docs.microsoft.com/en-us/xamarin/ios/internals/limitations (cit. on p. 20).

[32] P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns*. Ed. by Packt Publishing. Packt Publishing, 2017 (cit. on pp. 38, 39).

[33] React. *React Native*. url: https://reactnative.dev (cit. on p. 17).

[34] P. Saccomani. *Native Apps, Web Apps or Hybrid Apps? What's the Difference?* 2018. url: https://www.mobiloud.com/blog/native-web-or-hybrid-apps (cit. on pp. 15, 16).

[35] S. K. Sood, A. K. Sarje, and K. Singh. "Cryptanalysis of password authentication schemes: Current status and key issues". In: (2009) (cit. on p. 5).

[36] R. Soral. *React Native vs Iconic: Which Framework is best and why?* 2020. url: https://www.simform.com/react-native-vs-ionic/ (cit. on pp. 17, 18).

[37] S. Todavhich. "Advantages and disadvantages of Progressive Web Apps". In: (2019). url: https://moqod-software.medium.com/advantages-and-disadvantages-of-progressive-web-apps-6f019223cb17 (cit. on p. 16).

[38] TrustcomFinancial. *Online banking at your fingertips: mobile apps and the evolution of banking services*. 2020. url: https://trustcomfinancial.com/online-banking-evolution-mobile-apps/ (cit. on p. 1).

[39] I. Tzemos, A. P. Fournaris, and N. Sklavos. "Security and efficiency analysis of one time password techniques". In: (2016) (cit. on pp. 5, 6).

[40] F. Zhou. *Learn Objective-C: A Brief History*. url: https://www.binpress.com/objective-c-history/ (cit. on p. 24).

A

Survey Questionnaire

A lot of questions remain without answers about online security and new technologies emerge in order to protect users from several attacks. When it comes to handling sensitive data, companies tend to invest a lot on security in such way to make sure users' data remain private and can only be accessed and handled by authorized personnel, however, this investment, also depends on user's online behavior.

With this questionnaire, we intend to study user's online behavior, more specifically when it comes to Home Banking platforms.

Home Banking is a term that become popular since the 1980's and defines as a system whereby a person, at any given place and time, for as long as he/she has a device with internet connection, can access information about his/her bank account, make deposits, funds transactions and service or shopping payments.

A.1 Usage of Home-Banking Platforms

1. Do you use Home-Banking platforms?

    a) Yes

    b) No (skip to section A.3)

A.2 User's Behavior

You have answered "Yes" to the previous question about the usage of Home Banking platforms. In this question, we intend to collect information based on your online behavior on the devices you use to access these platforms.

Once you have answered this section, you may **ignore section A.2**.

1. Which device(s) do you use for online banking? You can select more than one option, if applied.

    a) Personal Computer (e.g., Laptop, Desktop)

b) Tablet or Smartphone

c) Public Computer (e.g., Library, Work)

2. Have you ever been victim of phishing or any kind of digital theft threats?
   Hint: Fraudulent attempt to obtain sensitive and confidential information such as usernames, passwords and credit or debit card details, through the disguise of a trusted entity in an electronic communication

   a) Yes

   b) No

3. In which of the following way(s) do you store your passwords? You can select more than one option, if applied

   a) I memorize them in my mind

   b) My Browser/Device stores them automatically

   c) In applications such as 1Password or Keychain

   d) On Physical paper, such as Post-its, papers or notebooks

   e) In the Notes App

4. Do you usually use different passwords for different platforms?

   a) Yes

   b) No

5. Do you use a unique password for your Home Banking platform?

   a) Yes

   b) No

6. Do you regularly change your password?

   a) Yes

   b) No

7. If you answered yes to the previous question, how often do you change your password?

   a) Regularly, at least every 30 days

   b) With some frequency, up to every 60 days

   c) Sometimes, up to every 90 days

d) Rarely, I only do such when I need to recover my password or the platform I want to access requires me to change my password

8. Which of the following procedures do you usually take after accessing your Home Banking service? You can select more than one option, if applied.

   a) Sign-out;
   b) Clear Cache;
   c) Clear History,
   d) Close the app without signing-out

9. Have you every accessed your Home Banking service outside your house?

   a) Yes
   b) No

10. If you answered yes to the previous question, how do you connect to the internet? You can select more than one option, if applied

    a) Cellular Data
    b) I search for a public network, such from a store or a friends' network

11. Do you keep your Operating System up-to-date with security patches?

    a) Yes
    b) No
    c) Don't Know

12. Do you have an anti-virus or anti-malware software installed on the devices you usually use to access your Home Banking service?

    a) Yes
    b) No
    c) Don't Know

13. Does your Bank sends you authorization tokens to proceed with online transactions? In case none of these options are applied, you can specify your Bank procedure

    a) Yes, via E-mail
    b) Yes, via SMS,
    c) No

d) Other. Please specify:

14. Does your Bank require a Matrix-Card every time you want to proceed with a online transaction?

    a) Yes

    b) No

A.3 You Don't Use Home-Banking Platforms

In case have answered **NO** to **Section A** first answer, whether you use *Home-Banking* applications or not, we would like to know which of the following(s) is the reason(s) why you do not use it.

In case that reason does not appear as an option, you can mention it on the last question.

1. What are your reason(s) for not using *Home-Banking* services?
   You can select more than one option

   a) I do not find them useful;

   b) I am afraid of phishing or different nature attacks

   c) My Bank does not offer *Home-Banking* solutions

   d) Other(s) reason(s)

2. If you have answered "**(d)** Other(s) Reason(s)"to the previous question, please specify:

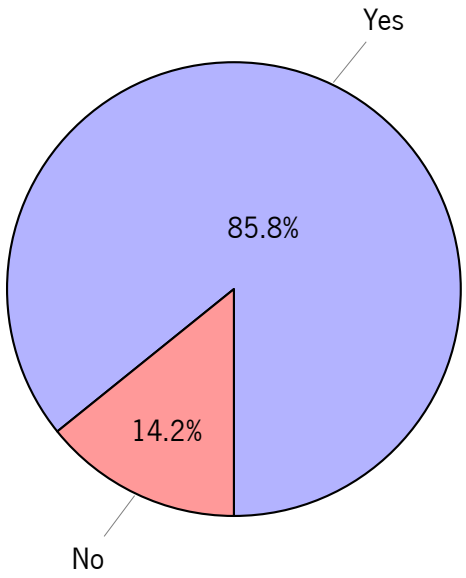
B

Survey Results

This section refers to the results obtained from the conducted survey under section A, where we have collected **answers from 134 different individuals**. 115 have said to use Home-Bank Applications, where only 19 have said not to use them.

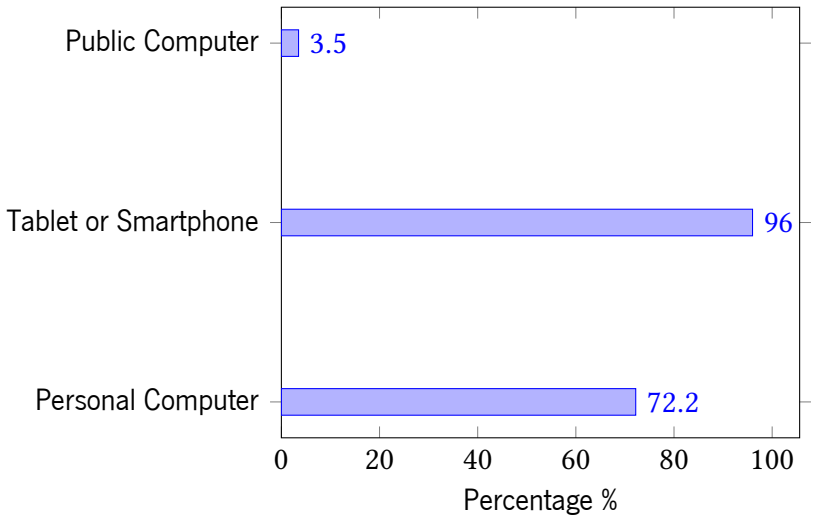
B.1 Usage of Home-Banking Platforms

1. Do you use Home-Banking platforms?



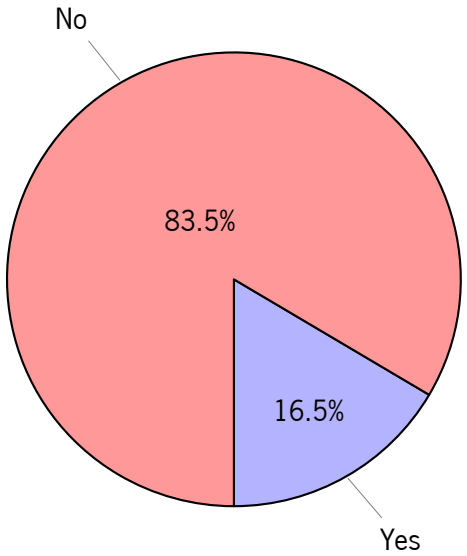
B.2 User's Behavior

1. Which device(s) do you use for online banking? You can select more than one option, if applied.

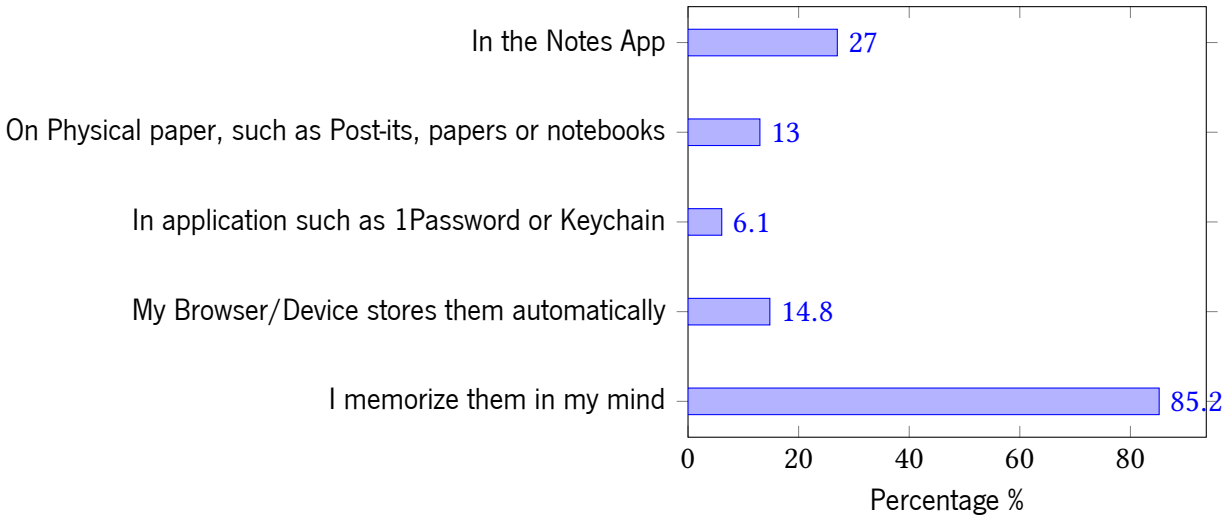


2. Have you ever been victim of phishing or any kind of digital theft threats?

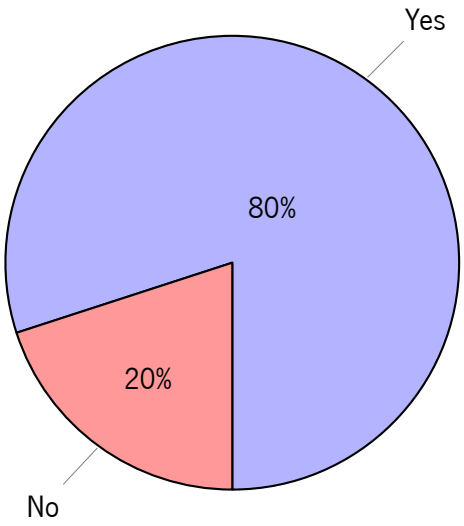
   Hint: Fraudulent attempt to obtain sensitive and confidential information such as usernames, passwords and credit or debit card details, through the disguise of a trusted entity in an electronic communication



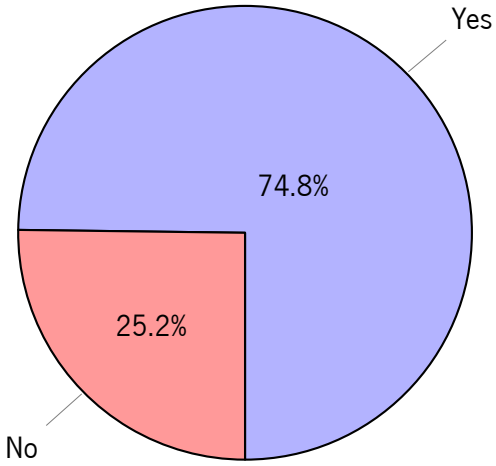
3. In which of the following way(s) do you store your passwords? You can select more than one option, if applied



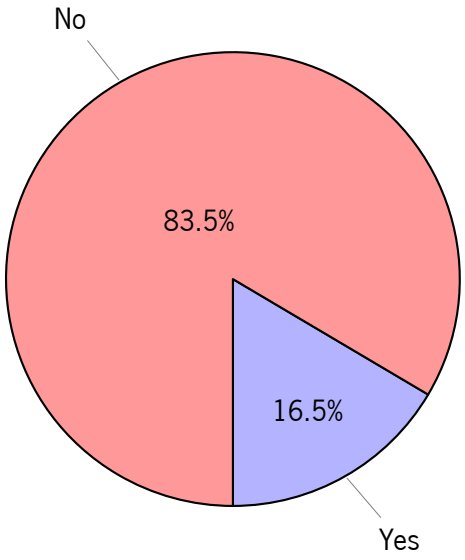
4. Do you usually use different passwords for different platforms?



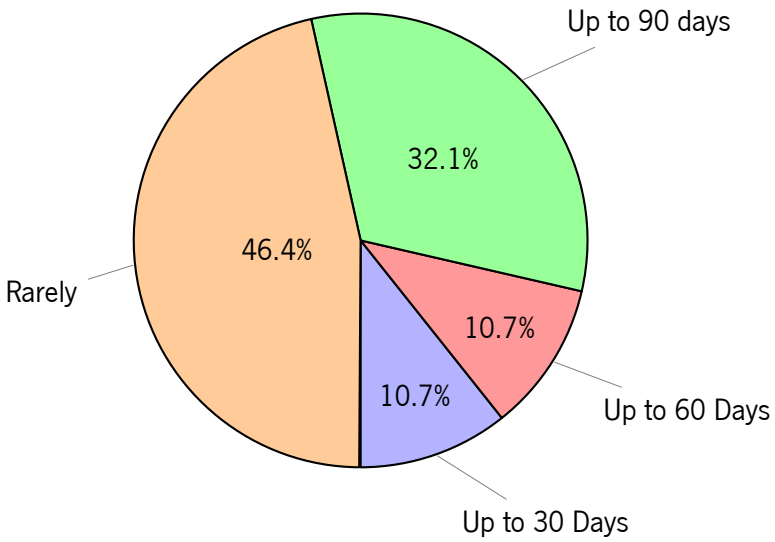
5. Do you use a unique password for your Home Banking platform?



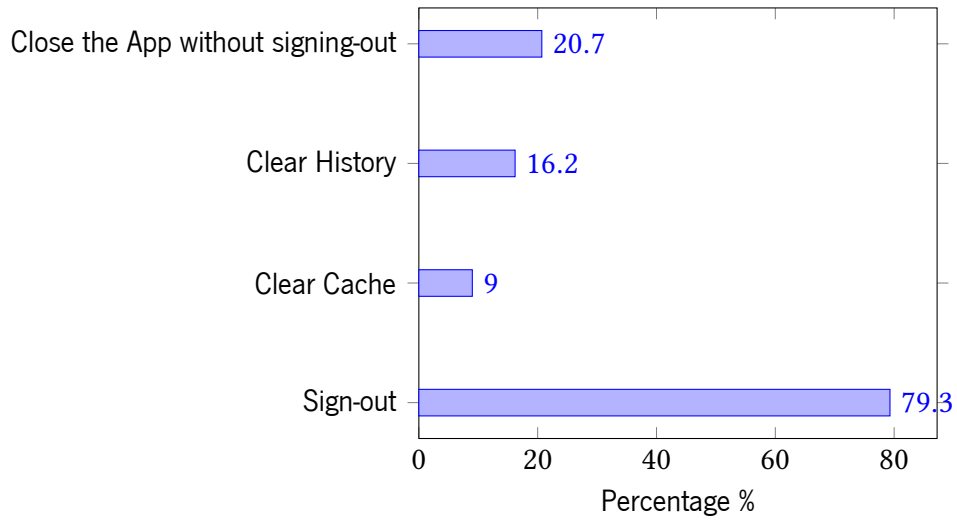
6. Do you regularly change your password?



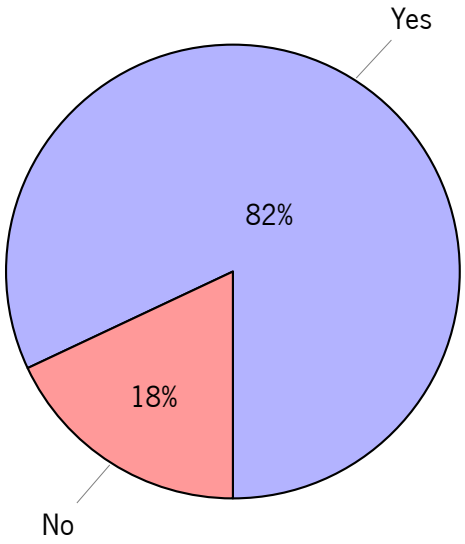
7. If you answered yes to the previous question, how often do you change your password?



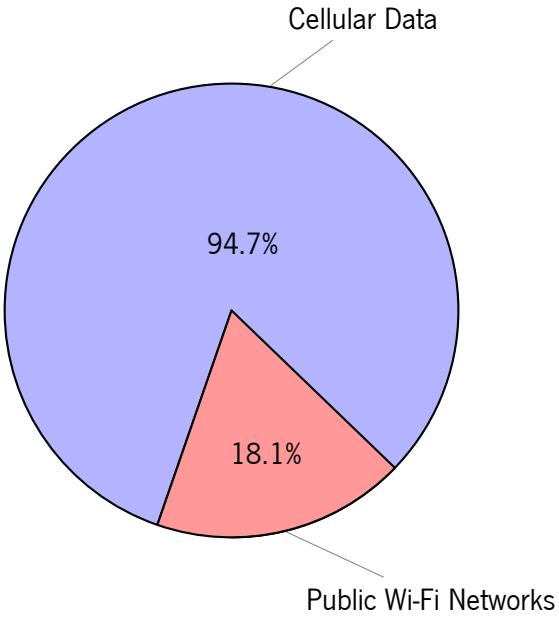
8. Which of the following procedures do you usually take after accessing your Home Banking service? You can select more than one option, if applied.



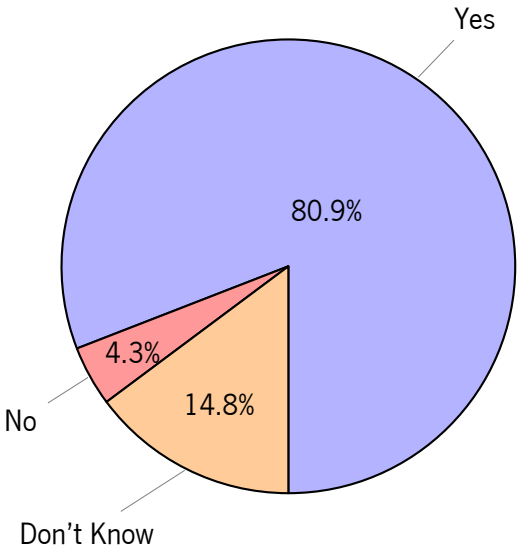
9. Have you every accessed your Home Banking service outside your house?



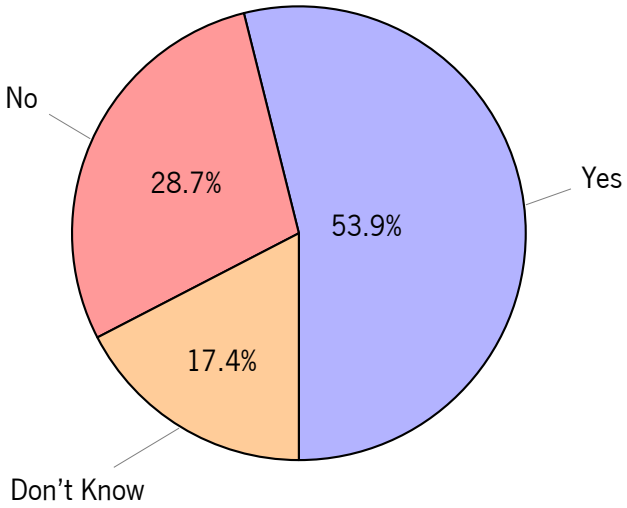
10. If you answered yes to the previous question, how do you connect to the internet? You can select more than one option, if applied



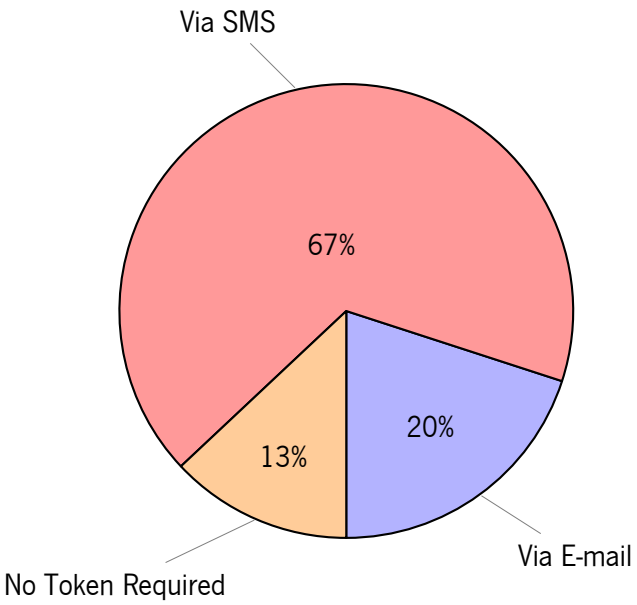
11. Do you keep your Operating System up-to-date with security patches?



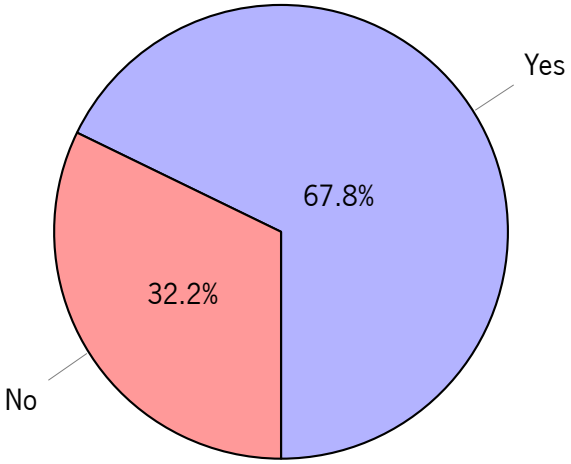
12. Do you have an anti-virus or anti-malware software installed on the devices you usually use to access your Home Banking service?



13. Does your Bank sends you authorization tokens to proceed with online transactions? In case none of these options are applied, you can specify your Bank procedure.

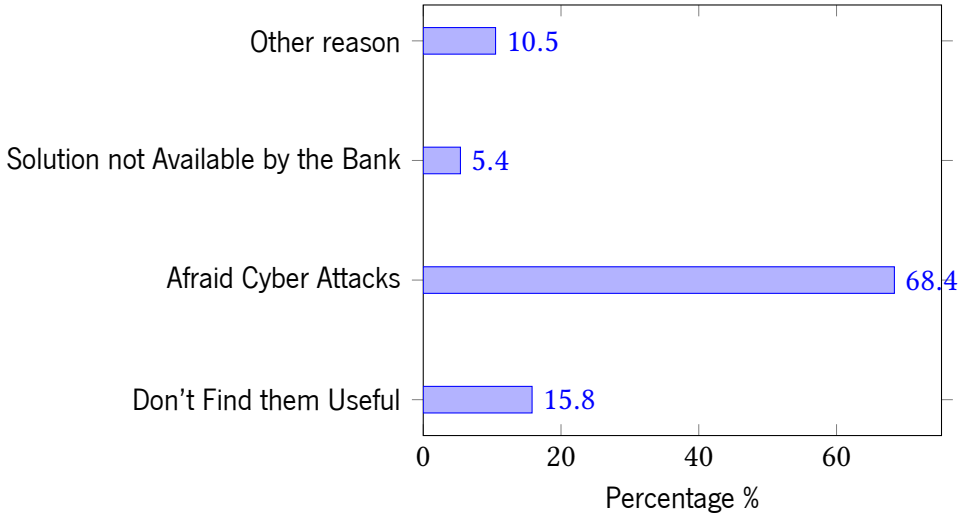


14. Does your Bank require a Matrix-Card every time you want to proceed with a online transaction?



B.3 You Don't Use Home-Banking Platforms

1. What are your reason(s) for not using *Home-Banking* services?
   You can select more than one option



2. If you have answered "**(d)** Other(s) Reason(s)"to the previous question, please specify: *These were the answers we have collected from the inquired individuals*

   a) Reliability

   b) I do not need them

   c) I'm afraid to use them wrongly and ending up at loss

   d) I have already been victim of Cyber attack

C

REST Interface

C.1 Register Code

GET	private/trustfactor/register-code

Output	{
	"QRCodeSize": "Number",
	"OperationId": "String",
	"CodeDuration": "Number",
	"QRCode": "String",
	"QRCodeURL": "qrCodeUrl",
	"DeeplinkURL": "deepLinkUrl"
	}

C.2 Create Transaction

POST	private/trustfactor/create-transaction

Input	{
	"OperationID": "String",
	"TransactionId": "Number",
	"TransactionData": "Object"
	}

Output	{
	"TransactionID": "String",
	"Duration": "Number" ,
	"Requirement": "Number"
	}

C.3 Refresh Register Code

POST	private/trustfactor/refresh-register-code

Input	{
	"OperationId": "String",
	"AppUniqueId": "String",
	"AppUsername": "String",
	"CodeOutputData": "Number"
	}

Output	{
	"QRCodeSize": "String",
	"CodeDuration": "Number",
	"QRCode": "String",
	"QRCodeURL": "String",
	"DeeplinkURL": "deepLinkUrl"
	}

C.4 Check Register Code

POST	private/trustfactor/check-register-code

Input	{
	"UniqueId": "String"
	}

Output	{
	"Status": "Number",
	"ErrorMessage": "String"
	}

C.5 Check Create Transaction

POST	private/trustfactor/check-create-transaction

Input	{
	"TransactionId": "String",
	"OperationId": "String",
	"Data": "String"
	}

Output	{
	"Status": "Number",
	"ErrorMessage": "String"
	}

C.6 Pending Authorization Transaction

POST	private/trustfactor/pending-authorization-transaction

Input	{
	"TransactionId": "String",
	"OperationId": "String",
	"Data": "String"
	}

Output	{
	"Success": "Boolean"
	}

C.7 Requirement

GET	private/trustfactor/requirement

Output	{
	"Requirement": "Number"
	}