

POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies



**Politecnico
di Torino**

Master's Degree Thesis

**Design and development of an enterprise
digital distribution platform for mobile
applications**

Supervisors

Prof. Sara VINCO

Ing. Ilario GERLERO

Ing. Claudio CHIEPPA

Candidate

Francesca DAVI

April 2022

Summary

Nowadays, over 6.3 billion smartphone users are present across the world and the mobile application industry is exploding. Given the success of this industry, more and more enterprises are embracing mobile technology and embedding enterprise mobile applications into their business, to support their work processes, manage the logistic flow and increase production efficiency.

Enterprise Mobile Applications (EMAs) are mobile apps developed by individual enterprises to support their jobs, required to run the enterprise itself. An EMA designed for a certain company is expected to be used by its own employees only. In this context, the digital distribution of enterprise mobile applications to final users, such as clients and partners, becomes a crucial topic.

The most popular way to distribute mobile applications is using the official *app stores* provided by Apple and Google. An app store is a digital distribution platform for mobile applications, allowing the users to browse and download them. Using public distribution platforms is fast, secure and the infrastructure is ready-to-use. On the other hand, public app stores may give access to restricted and confidential information not only to employees but also to unauthorized users.

The aim of the thesis is the development of an enterprise distribution platform for mobile applications, designed for a leading food company that needs to share its own enterprise applications with customers and partners in a private, direct and secure way, overcoming the privacy issue encountered with public app stores. The project was carried out in collaboration with Reply S.p.A., as part of a real industrial-oriented project.

The proposed solution aims at designing an architecture for distributing the enterprise mobile applications over-the-air (OTA), sharing public links that are valid only for a limited amount of time. Over-The-Air installation is direct and fast. Furthermore, limiting the validity time window of a link avoids that too many unwanted downloads could be performed.

In order to identify possible malicious behaviors, some analytical considerations were carried out to include security features by monitoring the downloads during the link validity time. In particular, the thesis considered the amount of downloads for a specific link and the geographical position of the final users, to prevent undue

spread of the applications. Additionally, downloaded data are aggregated to provide insight and average adoption of the applications.

The implemented solution is a web application composed of a Front-end and a Back-end, independently developed. The candidate was mainly involved in the Front-end development.

Figure 1 shows the main development phases and the way they are interconnected.

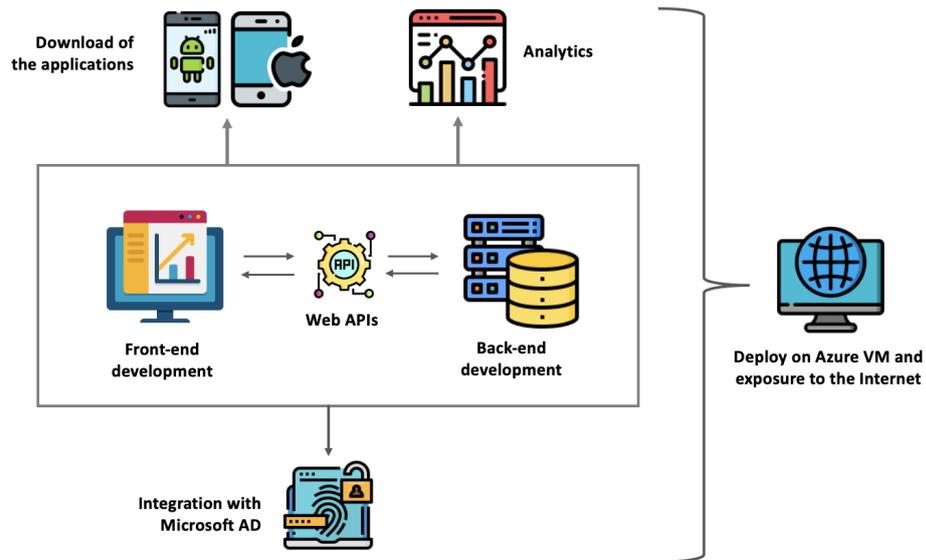


Figure 1: Methodology scheme

Front-end and Back-end were set up in parallel. The Front-end consists of everything the user interacts with, through a web browser. In accordance to the client company, five different pages were implemented to provide an interactive User Experience (UX). React JavaScript and Material-UI libraries were used to build the user interface. React was chosen in order to design a Single Page Application (SPA). Single-Page Applications are web applications built over just one single page and differ from traditional one because the content is loaded only once and subsequently only the parts getting new data are updated, communicating with the server through Web APIs. The complexity is slowly moved from the server to the client and most operations are executed in the browser. This approach makes the websites faster and responsive, since complete page reloads are unusual.

The Back-end is the part of the system hidden to the user and responsible of storing and manipulating the data. In the current architecture, a Web server along with a Microsoft SQLServer Database were implemented. The designed database is composed of several tables (**Application**, **Version**, **Link**, **Download**, **User** and

Groups), storing all data necessary to build the entire system.

Front-end and Back-end communicate through REST APIs, over the HTTP protocol. The Front-end makes *requests* to the Back-end, that sends back the desired data in JSON format as a *response*. This allows the designed web application to refresh just a single part of the page, as in all SPAs.

Once the structure of Front-end and Back-end was created, the focus was placed on the main functionalities of the system, i.e. the upload/download of the applications. First, the upload and storage to the platform of the `.apk` and `.ipa` files (the installation files of Android and iOS applications respectively) was designed. The Front-end side provides a form where the user can upload the intended file. When submitting it, a proper `POST` request is sent to the Back-end and the application is uploaded and saved to the platform.

After that, the download procedure was implemented. The Front-end side provides a form in which a link can be created by selecting the versions to be shared (one version per platform at most), and the validity time window. For Android devices, the download is the direct transfer of the file from the server to the device only. Once the download is completed, the user is asked to eventually install the application or just store the `.apk` file locally. The download procedure for iOS instead requires more effort. The direct transfer of the file from the server to the device only allows to locally store the `.ipa` file on the iOS device but not to start the installation process. The creation of a dedicated `manifest.plist` file, containing name, version, unique identifier of the application along with the URL for the download, is instead necessary to perform the installation.

As a next step, Azure Active Directory have been integrated to make the core functionalities of the system accessible to specific users only. Azure AD allows the developers to build applications in which users and customers can be authenticated through *Microsoft identity platform*, using their Microsoft work or social accounts. The integration with Azure AD, already in use at the client company, was performed using MSAL (*Microsoft Authentication Library*). MSAL enables developers to acquire tokens from the Microsoft Identity Platform in order to authenticate users. Two actors play a role in the system: the *authenticated user* and the *final user*. The authenticated user is able to access all the Front-end pages and to use all the designed functionalities. The final user instead receives from an authenticated user a public link, that allows to download a certain application during a limited period of time.

Once all the system was set up and made accessible to particular users only, the development of the analytical part took place. In this phase of the project, additional functionalities were added to the Front-end and Back-end already developed. The pages were populated with ad-hoc designed charts and maps containing data related to the amount and the position of the downloads. Thanks to these insights, the user can monitor the trend of the downloads of its own applications and identify possible

malicious behaviours, in order to preserve the secure sharing of the applications.

So far, all the system was locally developed only. After that both Front-end and Back-end were deployed on an Azure Virtual Machine supplied by the client company. Furthermore, a dedicated public domain was chosen and the designed Web Application was hosted on the Virtual Machine and exposed to the Internet through Microsoft IIS (*Internet Information Services*) configuration. After that, the designed system and the correlated features are available to the users through the Internet.

Acknowledgements

Grazie.

A mamma, papà e Giulia per la pazienza e le immense energie che hanno dovuto spendere in questi anni per sopportarmi. Per avermi spronata a non mollare mai e per avermi sempre supportato anche e soprattutto economicamente. Grazie.

A Dario, co-protagonista della fine della mia laurea triennale e dell'intero percorso magistrale. Per la sua comprensione infinita, per ogni singola notte prima degli esami passata silenziosamente accanto a me, per tutti i preziosi consigli che hanno contribuito a portarmi dove sono arrivata oggi. Grazie.

A Cinzia, per avermi designata come sua degna erede in tutto e per tutto da quel 28 maggio 1993. Per aver sempre risposto repentinamente alle mie folli richieste last minute di ripetizioni pre-esame e per aver spesso sentito più suo che mio questo lunghissimo percorso. Grazie Cy.

A Chiara, agli zii, a Patrizia e a ogni membro della mia famiglia, per non aver mai smesso di credere in me. Grazie.

A tutte le stelle che ho lassù, a quelli che non hanno vissuto abbastanza per essere qui con me oggi, ma che mi hanno insegnato i valori dell'umiltà e della determinazione, con cui ho affrontato questo percorso. Grazie.

A Giulia e Sonia, amiche di sempre, per aver condiviso con me gioie e dolori, per non avermi mai lasciata sola. Per avermi accompagnata durante tutti i periodi bui di questi anni. Grazie.

A tutta la Condove Che Conta, per "l'intrattenimento" (cit.) nei periodi di stress e per aver allietato la mia gioventù con le migliori serate. Grazie.

A Mila e Andrea, fedeli "custodi" di gran segreti durante il percorso triennale e sempre presenti durante tutte le battaglie affrontate lungo la Laurea magistrale. Siete i migliori compagni di università che potessi trovare. Grazie AmiciAmici.

Ai Chicks: Elena, Riccardo e Damiano. Per tutti gli esami preparati insieme, per le slides preparate fino a 2 minuti prima della presentazione, per i "bauli alzati". Grazie ragazzi, avete reso questo percorso magistrale più sereno. Speciale menzione va a Damiano, senza il quale tutta questa esperienza in Reply e questa tesi non sarebbero state possibili. Grazie amichetto.

A Sara Vinco, per la sua gentilezza e il suo sostegno durante la stesura di questo

elaborato. Grazie Professoressa.

A Claudio e Dante, per avermi presa per mano dal primo giorno di questa esperienza e aver creduto in me anche quando io stessa non ero in grado di farlo. Grazie.

A tutti i ragazzi della BU IIOT di Concept Engineering Reply. Per avermi fatta sentire a casa dal primo momento e aver reso questi mesi più leggeri e spensierati. Grazie ragazzi.

Al migliore amico a quattro zampe che si possa desiderare, Artù. Per avermi salvato la vita e le giornate in tante occasioni, senza nemmeno saperlo. Grazie mio eterno cucciolone.

Ad ogni singola persona che ho incontrato durante questo lungo percorso e che inconsapevolmente ha contribuito ad alleggerire il peso del Politecnico. Grazie.

E infine, grazie a ME. Per non aver essermi mai arresa, per avercela fatta. Per aver superato tutti gli ostacoli più o meno grandi che la vita mi ha messo davanti nel corso di questi lunghi anni. Grazie.

Table of Contents

List of Figures	XI
Acronyms	XIV
1 Introduction	1
2 State of the Art	4
2.1 TypeScript	4
2.2 Front-end	5
2.2.1 React	5
2.2.2 JSX/TSX	6
2.2.3 MUI	6
2.2.4 React Router	7
2.2.5 Axios	8
2.2.6 Recharts	8
2.2.7 React Simple Maps	8
2.2.8 React-i18next	8
2.3 Back End	9
2.3.1 NestJS	9
2.3.2 TypeORM	9
2.4 Azure AD	9
2.4.1 MSAL	10
2.5 IIS	10
3 Methodology Overview	12
4 Methodology	15
4.1 Front-end pages development	15
4.1.1 The Routes	16
4.1.2 The Dashboard	17
4.1.3 Applications Page	21

4.1.4	Application Details Page	22
4.1.5	Download Page	30
4.1.6	Profile Page	31
4.1.7	Multilanguage Support	36
4.2	WebServer and Database development	37
4.2.1	WebServer	37
4.2.2	Database	39
4.3	API	42
4.3.1	Front-end	42
4.3.2	Back-end	45
4.4	Download	46
4.4.1	Download for Android	47
4.4.2	Download for iOS	48
4.5	Microsoft Authentication Integration	50
4.5.1	Authenticated users	50
4.5.2	Final users	52
4.6	Analytics	53
4.6.1	Database update	53
4.6.2	Front-end components	53
4.7	Deployment on Azure Virtual Machine	57
4.7.1	Deployment	57
4.7.2	IIS configuration	59
5	Conclusions	61
	Bibliography	62

List of Figures

1	Methodology scheme	iii
2.1	TypeScript vs JavaScript	4
2.2	SPAs vs MPAs	5
3.1	Methodology scheme	12
4.1	Dashboard Page	18
4.2	<LastApps> component	19
4.3	<LastLinks> component	20
4.4	Applications Page	21
4.5	Applications Details Page: top-half page	22
4.6	Step 1	25
4.7	Step 2	26
4.8	Step 3	27
4.9	Step 4	27
4.10	Link Generation Procedure: Step 1	28
4.11	Link Generation Procedure: Step 2	29
4.12	Link Generation Procedure: Step 3	29
4.13	Download Page - Desktop OS	31
4.14	Profile Page	32
4.15	User instance creation - Step 1	33
4.16	User instance creation - Step 2	34
4.17	User instance creation - Step 3	34
4.18	Group creation - Step 1	35
4.19	Group creation - Step 2	35
4.20	Language menu selection in Application Bar	36
4.21	Download Procedure on Android [63]	48
4.22	Download Procedure on iOS [63]	49
4.23	<MapChart>	54
4.24	<DownloadBarChart>	55
4.25	<DownloadLinkChart>	56

4.26	Development instance	58
4.27	Production instance	58
4.28	New website creation with IIS	59
4.29	IIS - Rule Definition	60
4.30	IIS - Rules Overview	60

Acronyms

API

Application Programming Interface

EMA

Enterprise Mobile Application

JS

JavaScript

JSX

JavaScript XML

MPA

Multi-Page Application

MSAL

Microsoft Authentication Library

ORM

Object-Relational Mapping

OS

Operating System

OTA

Over-The-Air

RBMS

Relational Database Management System

SPA

Single-Page Application

TS

TypeScript

UI

User Interface

UX

User Experience

Chapter 1

Introduction

With the spread of smartphones in late 2000s, the impact of mobile applications faced a worth increase. Mobile applications consist of pieces of software/set of program that runs on a mobile device and perform certain tasks for the user [1]. The reason why mobile apps became so popular is that they provide services over tiny devices the user always has in its pocket.

Nowadays, over 6.3 billion smartphone users are present across the world and the mobile application industry is exploding [2].

Given the success of this industry, more and more enterprises are embracing mobile technology and enhancing their operations by embedding enterprise mobile applications into their business, to support their work processes, manage the logistic flow and increase production efficiency [3].

Enterprise Mobile Applications (EMAs) are mobile apps developed by individual enterprises for their employees, to support their jobs required to run the enterprise itself. An EMA designed for a certain company is expected to be used by its own employees only [4].

In in this context, the digital distribution of enterprise mobile applications to final users, such as clients and partners, becomes a crucial topic.

The term *digital distribution* refers to the delivery of digital media content (e.g. audio, video and software) without the use of physical media, normally by downloading it from the Internet [5].

The most popular way to distribute mobile applications is using the official *app stores* provided by Apple and Google. An app store is a digital distribution platform for mobile applications, allowing the users to browse and download them. Apple App Store was launched in July 2008 along with iPhone 3G and it was the first example of real app storefront [6]. Competitors came after, trying to emulate the same architectural idea. Apple App Store guarantees that more than two millions apps offered on that store are held to the highest standards for privacy, security, and content [7]. Google Play Store was released in October 2008 and it

distributes mobile applications for Android devices.

Using public distribution platforms is fast, secure and the infrastructure is ready-to-use. Some drawbacks are present, though. Mostly, enterprise mobile applications often contain private features devoted to employees only and this way they can be downloaded by unauthorized users. Furthermore, Apple takes some days to review an app uploaded to the store, to ensure reliability and privacy features [8] and at the end of the review period, the application can be rejected. The risk is that the release of an application devoted to internal company usage only is blocked because it does not respect the strict rules of Apple.

The aim of the thesis is the development of an enterprise distribution platform for mobile applications, designed for a leading food company that needs to share its own enterprise applications to customers and partners in a private, direct and secure way, overcoming the drawbacks encountered with public stores. The proposed solution aims at designing an architecture for distributing the applications over-the-air (OTA), via public links that are valid only for a limited amount of time. Over-The-Air installation is direct and fast. Furthermore, limiting the validity time window of a link avoids that too many unwanted downloads can be performed. In order to identify possible malicious behaviors, some analytical considerations were carried out to include security features by monitoring the downloads during the link validity time, like the amount of downloads for a specific link and the geographical position of the final users, to prevent undue spread of the applications. Additionally, the downloaded data are aggregated to provide insight and average adoption of the applications.

The project was carried out in collaboration with Concept Engineering Reply, as part of a real industrial oriented project. Concept Engineering Reply covers the software development process from the Device, through the Gateway, to the Cloud or Smartphone App. Its customized software typically handles a few thousand of devices, covers security and maintainability. Concept Engineering Reply guides customers through the design process of their application, specify the ideal Cloud-solution architecture, develop in agile teams. Together with partners for hardware and telecommunication it provides to the customers an individual software solution based on IoT requirements.

This thesis is organized into these sections:

- The *State of The Art* in section 2 presents a quick overview of all the technologies applied to build the intended solution. For each of the library involved and further cited along the work, a short description is reported.
- The *Methodology Overview* in section 3 briefly presents the methodology applied to develop the required platform. The main development phases are highlighted and described. A scheme clarifying the logical interconnections among them is reported too.

- The *Methodology* in section 4 is the core of the work. It describes all the phases introduced in section 3 in details, with reference to the technologies introduced in section 2.
- The *Conclusion* in section 5 summarizes the work and presents possible further improvements available.

Chapter 2

State of the Art

2.1 TypeScript

TypeScript (TS) was chosen as a programming language to develop both Front-end and Back-end. TypeScript is an open-source programming language developed by Microsoft and it is a superset of JavaScript (JS), because it adds some additional features to JS, as shown in Figure 2.1.

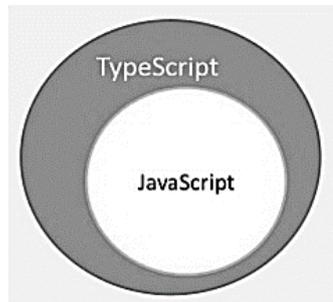


Figure 2.1: TypeScript vs JavaScript

All the JS code is also valid for TypeScript [9]. It shares the syntax and run-time behaviour with JS and supports all the JS libraries. Since browsers are not able to read TS code, TS files first have to be *transpiled* into JS before the execution. TypeScript associates an explicit data type to each variable. Using a typed programming language prevents from possible coding errors and bugs without deploying specific lines of code [10].

TypeScript supports the same *primitive* types JavaScript uses: `string`, `number`, and `boolean`. Developers can create their own variable types as well, in order to refer to the same data structure more than once with a single name [11].

2.2 Front-end

In this section all the libraries adopted to build the user interface (UI) are reported and described.

2.2.1 React

React is a JavaScript library for building user interfaces for Single-Page Applications (SPA). The traditional web approach provides the use of Multi Page Applications (MPA), that refresh the entire page when its content changes. Single-Page Applications instead are web applications built over just one single page and differ from traditional one because the content is loaded only once and subsequently only the parts the user interacts with are updated, communicating with the server through web APIs. The complexity is slowly moved from the server to the client and most operations are executed in the browser. This approach makes the websites faster and responsive, since complete reloads of the pages are unusual [12].

Figure 2.2 shows SPAs and MPAs different approaches.

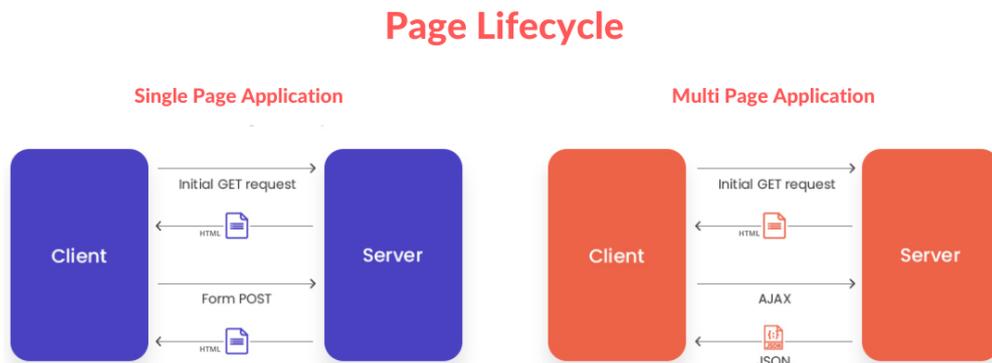


Figure 2.2: SPAs vs MPAs

React is one of the main libraries for building SPAs. It is a component-based library, meaning that the user interface consists of several independent and reusable components accordingly combined [13]. Every single component is able to keep and modify its own internal state, making possible to re-render the web page without reloading the entire content.

React *Hooks* are special functions that lets the developer “hook into” React features. They allows to use state and other React features without writing a class. `useState` and `useEffect` are two main Hooks, deeply used in thesis development. `useState` declares a *state variable*. and then allows to preserve some values between the function calls. It returns a pair of values: the current state and a function that updates it [14].

`useEffect` instead allows the developers to perform side effects in function components. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects [15].

2.2.2 JSX/TSX

JavaScript-XML (JSX) is a syntax extension of JS that allows to write HTML in React and makes creating React components easier. To create a React component, it is sufficient to define a JS function returning one or more HTML elements combined [16]. The following lines of code contain the definition of a simple React component that will render a HTML text on a web page.

Listing 2.1: JSX component Example

```
1   export default unctio HelloWorld() {  
2     return <h1>Hello World!</h1>;  
3   }
```

Once developed, the same component can be reused in different web pages just importing it, typing "`import HelloWorld from ...`". It is possible to nest different HTML elements by wrapping them in a single `<div>` HTML element.

A TSX file is just a TypeScript (TS) file written using JSX syntax. Each variable involved needs an explicit data type declaration. The following approach was adopted for the entire development of the Front End: a set of React components written in TSX are combined to build all the pages, further described in section 4.1.

2.2.3 MUI

Material-UI (MUI) is another library employed in the project to build the user interface. It is a React library providing a set of pre-built and customizable components that allow the developers to speed up the UI creation process [17]. MUI brings together two of the main technologies for front end development: it exposes Google Material Design principles in Facebook React. It is a bridge between developers and designers worlds [18]. A list of the main pre-built components used to design the UIs, along with a short explanation, is reported below.

- **Autocomplete**: a text input with a panel of suggested options [19]. So the value for the textbox has to be chosen among a predefined set of values.
- **Button**: a button allows the users to perform actions and make choices [20]. They can appear under the form of text or icons.
- **Card**: a small surface displaying content and actions about a single topic [21].
- **Chip**: a small element that represents an input, an action or a message [22].
- **Dialog**: a type of modal window that appears to provide critical information or ask for a decision [23].
- **Progress**: an indicator that informs users about the status of ongoing processes [24].
- **Select**: it is used to collect user provided information from a list of options [25]. Adding the `multiple` property, more than one element from the list can be chosen.
- **Stepper**: it displays progress through a sequence of numbered steps [26].
- **Table**: it displays information in an organized way [27]. It is composed of several `<TableRow>` elements which in turns are composed of smaller `<TableCell>`s.
- **Textfield**: it allows users to enter some text [28].

2.2.4 React Router

React Router is a library to enable routing in React web applications. It allows the navigation between different pages, changing the URL. *Routing* is the process of deciding which React elements will be rendered on a given page of the web application and how they will be nested [29].

To use React Router, a router element has to be rendered in the root of the element tree. `<BrowserRouter>` or `<HashRouter>` routers should be used in a web browser. Both store the current location in the browser address bar, but the former uses clean URLs, while the latter hashes the URLs [30]. Inside the router component, the *routes* have to be declared. When using JSX/TSX, a `<Routes>` wrapper and a `<Route>` element or each single page have to be stated. A `<Route>` takes two parameters: a `path` and an `element` to be rendered when the URL matches the path [31].

The library also provides a set of Hooks, to ease the navigation between different views. `useNavigate` hook is a function allowing to programmatically navigate to another page, for example after a form submission [32].

`useParams` hook returns some key/value pairs of the dynamic *params* from the current URL. It allows the developer to save params in state variables and exploit them [33].

2.2.5 Axios

Axios is a JavaScript library used to send asynchronous HTTP **Requests** to REST endpoints and perform the CRUD (Create, Read, Update, Delete) operations. To do that, it already provides methods like `.get()`, `.post()`, `.put()` and `.delete()` [34]. The reason why using Axios is to get support for *request* and *response* interception and for the conversion of received data into JSON format.

In React applications, the developer often needs to get data from the external sources. Axios helps in retrieving the data adding it to the state [35]. This library can be used both in server-side application and in the browser.

2.2.6 Recharts

Recharts is a Redefined Chart library, helping the developer to easily build charts in React applications [36]. Recharts provides a set of pre-built components, containing charts of various types. One only needs to import the desired chart (e.g. `<BarChart>`), passing the data to be represented as a parameter and set the label on the axis, the legends and the scale using the already present child components [37].

2.2.7 React Simple Maps

React Simple Maps is a React library allowing to build SVG maps in React. This library exposes a set of pre-built components that can be combined to create SVG mapcharts with markers and annotations. To render a map, a reference to a valid `topojson` file has to be provided [38]. Three main components are used to design the map. `<ComposableMap>` component is a wrapper component for every mapchart built with this library [39]. The dimensions and a particular projection can be passed as a parameter. Inside the wrapper, the `<Geographies>` component allows to fetch geographies files. A link to a valid `geojson` file or directly a `geojson` array can be set as a parameter for this component [40]. `<Geographies>` is a wrapper for `<Geography>` component, able to render geographic shapes as SVG paths [41].

2.2.8 React-i18next

React-i18next is an internationalization library for React applications. *Internationalization* is an important feature for a web application, allowing to overcome the language barrier among people who use the same software [42].

This library provides a set of components to ensure that needed translations of strings get loaded or that the right content gets rendered when the language changes [43]. React-i18next is also scalable: it allows to separate translations into several files and to load them on request.

2.3 Back End

This section focuses on the libraries adopted to build the Back-end side of the system.

2.3.1 NestJS

Nest (NestJS) is a framework for building efficient and scalable Node.js server-side applications. It is built with and fully supports TypeScript. NestJS provides a higher level of abstraction above the common Node.js frameworks (like Express), but also directly exposes their APIs to the developer. This allows the developers to use all the third-party libraries available for those platforms [44].

When dealing with server-side applications, NestJS is a good choice since because it is built over a few simple components (controllers, modules, and providers), that help dividing the app structure into microservices. *Controllers* handles incoming requests and return responses to the client-side. *Modules* helps to organize the structure of the application, separating the functionalities into reusable pieces. *Providers* abstract the application logic away from the user [45].

2.3.2 TypeORM

TypeORM is a TypeScript ORM (Object-Relational Mapper) library that connects a TypeScript application to a relational database. An ORM is a programming technique used to convert data between Relational Databases and Object-Oriented Programming languages (OOP) [46]. It allows to write queries without writing explicit SQL queries.

TypeORM helps to develop any kind of application that uses databases, from small applications with a few tables to large scale enterprise applications with multiple databases [47]. It works well with `Nest.js`, already cited in previous section.

2.4 Azure AD

Azure Active Directory (Azure AD) is a centralized cloud-based identity and access management service, that provides single sign-on and multifactor authentication

[48]. This service helps company employees to access external resources (such as Microsoft 365, the Azure portal...), and thousands of other applications. Azure AD also helps them to access internal resources like apps on the company network, along with any cloud apps developed by their own organization [49].

Azure AD is intended also for application developers that can use it to add authentication process to their applications, allowing it to work with a users' pre-existing credentials, avoiding to design an authentication architecture from scratch.

Azure AD allows the developers to build applications in which users and customers can be authenticated through *Microsoft identity platform*, using their Microsoft work or social accounts [50]. Authorized access to specific designed APIs can be provided using Microsoft identity platform, though.

The *Authentication* is the process of proving that a user is who he says he is. The *Authorization* is the act of granting an authenticated party permission to do something. It specifies the data a user is allowed to access and what he can do with that data [51].

Microsoft identity platform can be integrated into a Web Application using MSAL Library, quickly described in the next section 2.4.1.

2.4.1 MSAL

The *Microsoft Authentication Library* (MSAL) enables developers to acquire tokens from the Microsoft identity platform in order to authenticate users and access secured Web APIs [52].

MSAL can be used in many application scenarios. Microsoft Authentication Library for JavaScript (MSAL.js) enables both client-side and server-side JS applications to authenticate users using Azure AD for work and school accounts, Microsoft personal accounts (MSA), and social identity providers accounts [53].

MSAL for React (`@azure/msal-react`) is a React-oriented package to enable authentication in Javascript Single-Page Applications [54]. It provides a set of pre-built React components to easily implement the authentication process in React applications.

2.5 IIS

Internet Information Services (IIS) for Windows Server is a flexible, secure and manageable Web server for *hosting* web applications on the Web [55].

An IIS Web server accepts requests from remote devices and returns the intended response. This basic functionality allows web servers to share and deliver information across the Internet [56].

IIS provides a scalable architecture. Instead of keeping the majority of functionality within the server itself, IIS include a Web server engine in which several components, called modules, can be added or removed, depending on the developer needs. Modules are individual features that the server uses to process requests [57].

URL Rewrite Module has to be cited, since it was used in the project. The Microsoft URL Rewrite Module enables server administrators to create customized rules to map request URLs to other URLs easier to remember of to perform redirects and send custom responses [58].

Chapter 3

Methodology Overview

The goal of this thesis is the development of an enterprise distribution platform for mobile applications, for a leading food company that wants to share its own enterprise applications to customers and partners in a private and secure way.

The developed solution allows to distribute the applications over-the-air via public links that are valid only for a limited amount of time.

This chapter quickly describes the methodology applied in the thesis, built by using the technologies introduced in section 2.

Figure 3.1 shows the scheme of the methodology adopted to solve the problem.

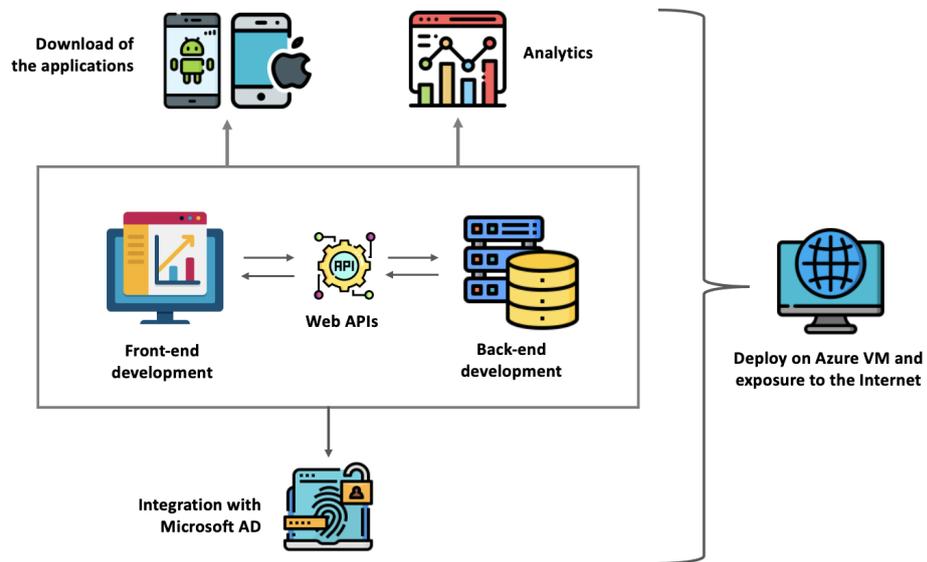


Figure 3.1: Methodology scheme

The proposed solution is a web application composed of a Front-end and a Back-end, developed independently.

The Front-end consists of everything the user interacts with, through a web browser. React JavaScript and MUI libraries were used to build the user interface. According to the client company, five different pages were implemented to provide an interactive User Experience (UX). As mentioned in section 2.2.1, React is used to design Single Page Applications (SPAs), so React Router library was necessary to allow the navigation among different views. Front-end development will be described in details in section 4.1.

The Back-end is the part of the system hidden to the user and responsible of storing and manipulating the data. In the current architecture, a Web server along with a Microsoft SQLServer Database were implemented. The designed database is composed of several tables (**Application**, **Version**, **Link**, **Download**, **User** and **Groups**), storing all the data necessary to build the entire system. The Back-end development will be further explained in section 4.2.

Front-end and Back-end communicate through REST APIs, over the HTTP protocol. The Front-end makes *requests* to the Back-end, that sends back the desired data in JSON format as a *response*. This allows the designed web application to refresh just a single part of the page, as in all SPAs. A more detailed description of the APIs will be provided in section 4.3.

Once the structure of Front-end and Back-end was created, the focus was placed on the main functionalities of the system, i.e. the upload/download of the applications. First, the upload and storage to the platform of the **.apk** and **.ipa** files (the installation files of Android and iOS applications respectively) was designed. The Front-end side provides a form where the user can upload the intended file along with some information. When submitting it, a proper **POST** request is sent to the Back-end side and the application is uploaded to the platform.

After that, the download procedure was implemented. The Front-end side provides a form in which the versions to be shared (one version per platform at most), and the validity time window of the link can be selected. For Android devices, the download is the direct transfer of the file from the server to the device only. Once the download is completed, the user is asked to install the application or just store the **.apk** file locally.

The download procedure for iOS instead requires more effort. The direct transfer of the file from the server to the device only allows to locally store the **.ipa** file on the iOS device but not to start the installation process. The creation of a dedicated **manifest.plist** file, containing name, version, unique identifier of the application along with the URL for the download, is instead necessary to perform the installation. The download procedures and their differences will be described in details in section 4.4.

To make the core functionalities of the system accessible to specific users only,

the integration with the Microsoft Identity Platform, already employed by the client company, was performed using MSAL (*Microsoft Authentication Library*) [50]. MSAL enables developers to acquire tokens from the Microsoft Identity Platform in order to authenticate users.

Two actors play a role in the system: the *authenticated user* and the *final user*. The authenticated user is able to access all the Front-end pages, to upload the mobile applications to the platform and to generate the temporary links for the downloads. The final user instead receives from an authenticated user a public link that allows to download a certain application during a limited period of time. The users management phase will be further discussed in section 4.5.

Once all the system was set up and made it accessible to particular users only, the development of the analytical part took place. In this phase of the project, additional functionalities were added to the Front-end and Back-end already developed. The pages were populated with ad-hoc designed charts and maps containing data related to the amount and to the position of the downloads. These statistics differ from one user to another. Thanks to these insights, the user can monitor the trend of the downloads of its own applications and identify possible malicious behaviours, in order to preserve the secure sharing of the applications. A more detailed description of the statistics implemented will be reported in section 4.6.

So far, all the system was locally developed only. After that, both Front-end and Back-end were deployed on an Azure Virtual Machine supplied by the client company. Furthermore, a dedicated public domain was chosen and the web application was exposed to the Internet through Microsoft IIS (*Internet Information Services*) configuration. Thus at that moment, the designed system and the correlated features are available to the users through the Internet. More information about the deployment to the Internet of the web application will be presented in section 4.7.

The last phase of the development is a test session of all the implemented functionalities, which is still ongoing. As soon as the system will be delivered to the client company, more insights about the system performances will be available.

Chapter 4

Methodology

This chapter aims at explaining every phase of the project development, introduced in section 3, in details. The first phase provides the Front-end and Back-end development. Both are coded using TypeScript as a programming language, quickly described in section 2.1. They communicate through REST API and the Front-end receives the data to be shown in JSON format. Most focus will be put on the Front-end development part which is the one where the candidate was mainly involved.

4.1 Front-end pages development

The Front-end development started with the creation of a React application, introduced in section 2.2.1. First, to create a React app, NodeJS is required to be installed. Then, a directory containing all the structure of the code and all the necessary dependencies is created running the terminal command `npm create-react-app my-app --template typescript`. The project folder should contain necessarily the following files:

- `public/index.html` that is the page template;
- `src/index.js` that is the JavaScript entry point;

`index.js` renders the `<App>` root component, which in turns encapsulates all the other designed components.

According to the client company, five different pages were created:

1. The **Dashboard**, containing the last applications uploaded to the platform and the last links generated for the download, along with an overview of the statistical analyses further explained in section 4.6.

2. The **Applications Page**, containing the list of all the applications uploaded to the platform.
3. The **Application Details Page**, showing all the details related to a single application (versions, generated links and statistical analysis of its adoption).
4. The **Download Page**, which is a public page allowing final users to download the applications.
5. The **Profile Page**, containing all the information related to users and groups.

Every page was written in TypeScript and the UI was built with the aid of Material-UI components, quickly introduced in section 2.2.3. Using encapsulated components helps programming in a good modular way.

4.1.1 The Routes

Each page must have a unique route to allow the navigation between different views in React Single-Page applications. The routes were implemented using *React Router* library, introduced in section 2.2.4. First of all, a *router* element has to be rendered. This was performed by declaring a `<BrowserRouter>`, which stores the current location in the browser address bar using clear URLs. Inside the `<BrowserRouter>`, the `<Routes>` were declared and one single `<Route>` was associated to each page. A `<Route>` takes a `path` and a JSX/TSX `element` to be rendered as parameters.

The following lines of code show how the routes, defined to allow the navigation, were declared.

Listing 4.1: Routes of different pages

```
1  function Pages() {
2    return (
3      <div>
4        <BrowserRouter>
5          <Routes>
6            <Route path="/" element={<Dashboard />}></Route>
7            <Route path="applications"
8              element={<ApplicationsPage />}>
9              </Route>
10           <Route
11             path="applications/:appID"
12             element={<ApplicationDetailsPage />}
13           >></Route>
14           <Route
15             path=":appID/download/:uuid"
16             element={<DownloadPage />}>
17           </Route>
```

```

18         <Route path="profile" element={<ProfilePage />}></
Route>
19         <Route path="*" element={<PageNotFound />}></Route>
20     </Routes>
21     </BrowserRouter>
22 </div>
23 );
24 }

```

The `<Dashboard>` having `"/` as a path is the homepage loaded when authenticating to the system. The `<ApplicationsPage>` element is rendered when navigating to the `"/applications"` path and the `ProfilePage` element is shown when navigating to `"/profile"` path.

Some URLs contains variable parameters. Syntactically, they are indicated as colon followed by the parameter, e.g. `:parameter`. This is the case of the URLs for `<ApplicationDetailsPage>` and `<DownloadPage>`, where different values of the parameters imply different element rendering. When navigating to `"applications/:appID"`, only the specific information related to the `:appID` application are rendered.

Finally, when a user tries to access a route matching no one of the reported paths, a `<PageNotFound>` element is rendered. This is performed by indicating the path as `"*"`. Once the routes were prepared, the real development of the pages was taken in charge. The next sections will focus on the design of each page in details.

4.1.2 The Dashboard

The Dashboard is composed of a MUI `<Grid>` containing encapsulated components, visible in the following lines of code.

Listing 4.2: Dashboard components

```

1 function Dashboard() {
2   return (
3     <div className="Dashboard">
4       <Grid container spacing={3}>
5         <Grid item xs={12} md={6} lg={6}>
6           <LastApps reverseData={reverseApps}/>
7         </Grid>
8         <Grid item xs={12} md={6} lg={6}>
9           <LastLinks reverseData={reverseLinks} />
10        </Grid>
11        <Grid item xs={12} md={4} lg={4}>
12          <MapChart countries={newObject4Plot}/>
13        </Grid >
14        <Grid item xs={12} md={4} lg={4}>
15          <DownloadBarChart downloads={download4Plot}/>

```

```

16     </Grid>
17     <Grid item xs={12} md={4} lg={4}>
18       <DownloadLinkChart downloads={Link4Download}/>
19     </Grid >
20   </Grid>
21 </div>
22 );
23 }

```

<Grid> helps organizing the UI and building responsive layouts, adapting the pages to the screen size of the devices. The <Grid item xs=12 md=6 lg=6> line of code makes the components occupy the entire screen (12/12) in small devices and half of the screen (6/12) in large devices.

Each of the five reported components is a Material-UI <Card> component. A <Card> is a small surface that can contain text and images. Figure 4.1 shows the quick overview provided by dashboard page.

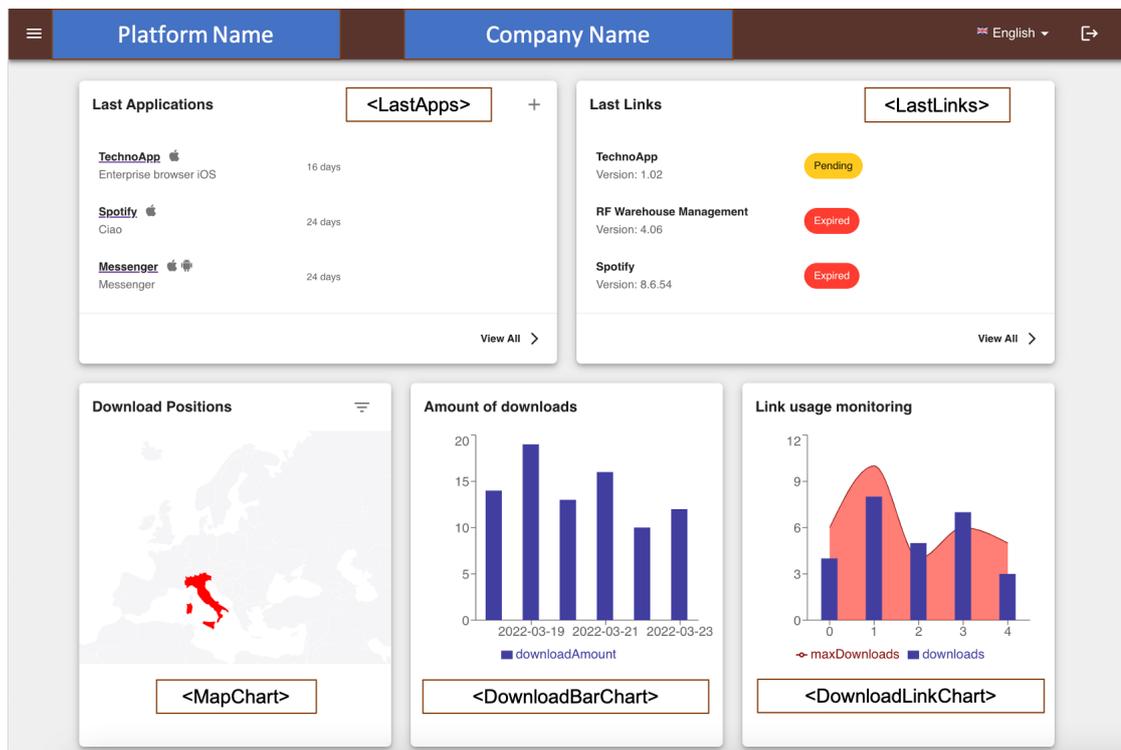


Figure 4.1: Dashboard Page

The top-left <Card> reports the last applications uploaded to the distribution platform. Figure 4.2 shows the <LastApps> component.

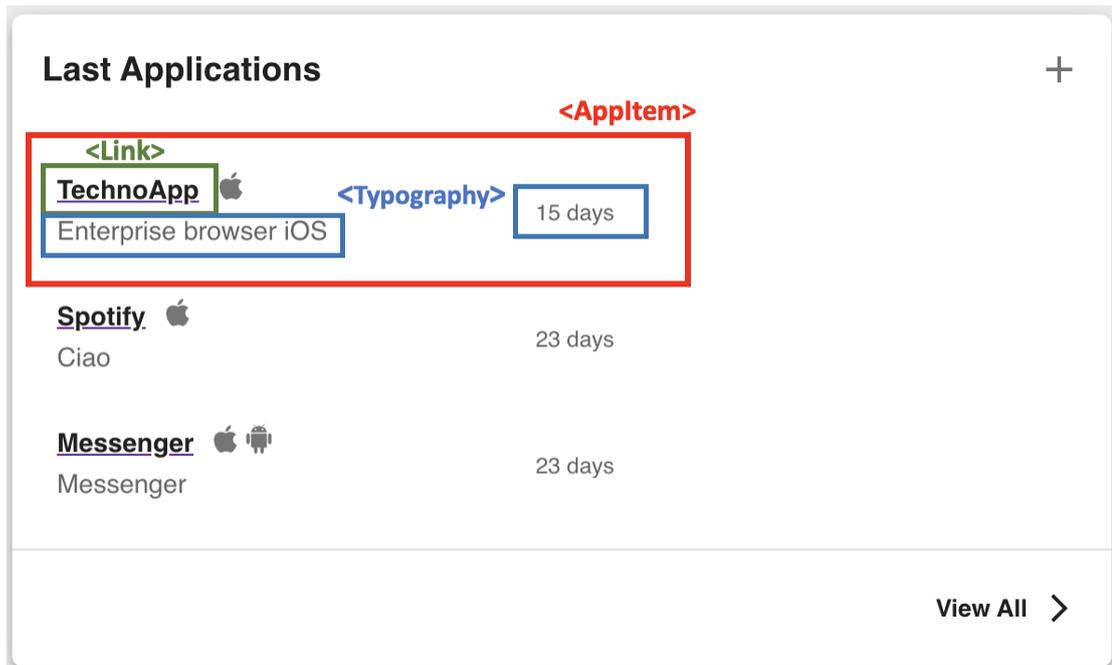


Figure 4.2: `<LastApps>` component

For each application, an `<AppItem>` component was created. It is a combination of `<Link>` React Router and `<Typography>` MUI components. The former allows to navigate from the home page to the one containing the details related to a single application, by clicking on the name. The latter reports the description and the upload date of the application.

The top-right `<Card>` renders the last links to be shared with customers or partners for the download of the applications. Similarly to before, a `<LinkItem>` was created for each generated link. It is again the combination of `<Link>` React Router and `<Typography>` MUI. Furthermore, a reusable ad-hoc `<LinkProgress>` component was built to monitor the progress of the validity of the link. To set it up, the *conditional rendering* approach of React was adopted: different components are rendered based on different conditions expressed by JS *if*.

Listing 4.3: `<LinkProgress>` component

```

1  <Box>
2    {progress === 0 && <Chip label="Pending" variant="filled "
3    color="warning"></Chip>}
4    {progress === 100 && <Chip label="Expired" variant="filled "
5    color="error"></Chip>}
6    {(progress > 0 && progress < 100) &&
  <LinearProgress
    variant="determinate"

```

```

7     value={progress}
8     color="primary"
9     sx={{width:160, height: 8, borderRadius: 10}} />}
10  </Box>

```

As the lines of code reported in 4.3 show, different MUI elements are rendered based on the value of the `Progress` variable, which is a **number** between 0 and 100, whose value is computed as the percentage of time already elapsed between the `validity_start` and `validity_end` of a link. This value is recalculated every second, so that the progress bar can be continuously updated, providing a more interactive UI. Figure 4.3 shows the three different possibilities the user can face, dealing with a link. Meanwhile the generated link is valid, a continuously updated progress bar is shown; once the link expires or it is still not valid, a "Expired" or "Pending" message appears.

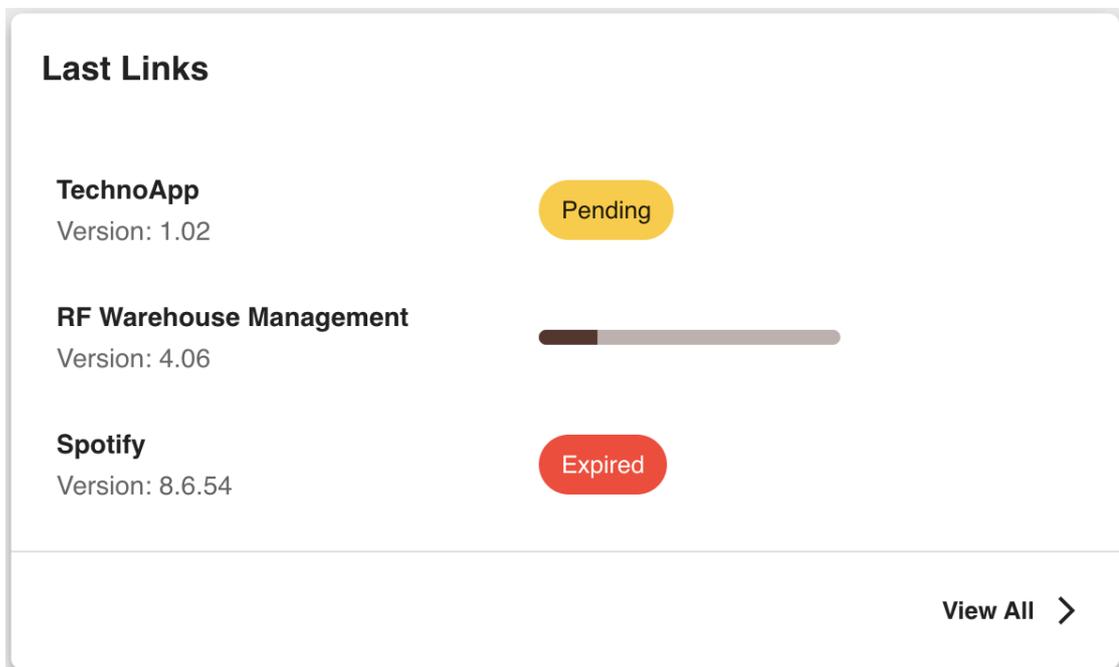


Figure 4.3: `<LastLinks>` component

The bottom `<Card>`s are related to the analytical part. The bottom-left `<Card>` shows the `<MapChart>` component, a Geographic Heat Map, created with *React Simple Maps* library, quickly introduced in section 2.2.7.

The bottom-centered `<Card>` contains the `<DownloadBarChart>` component, a chart showing the trend of the total downloads of its own applications during the last two weeks, built using *Recharts* library, described in section 2.2.6. This allows the authenticated user to see the total amount of downloads for its applications,

aggregated per apps.

The bottom-right one contains the `<DownloadLinkChart>` component, a chart showing the total amount of downloads per link. The last three components are further described in section 4.6.

The data to be shown are fetched from an API endpoint using the React Hook `useEffect`, able to perform side effects in components after rendering. The GET/POST requests are performed through Axios library, described in section 2.2.5. Further details about REST API and about the way the data are taken will be presented in section 4.3.

4.1.3 Applications Page

The *Applications Page* shows a single Material-UI `<Table>` containing all the applications uploaded to the platform by the current user. Figure 4.4 shows the designed UI for Applications Page.

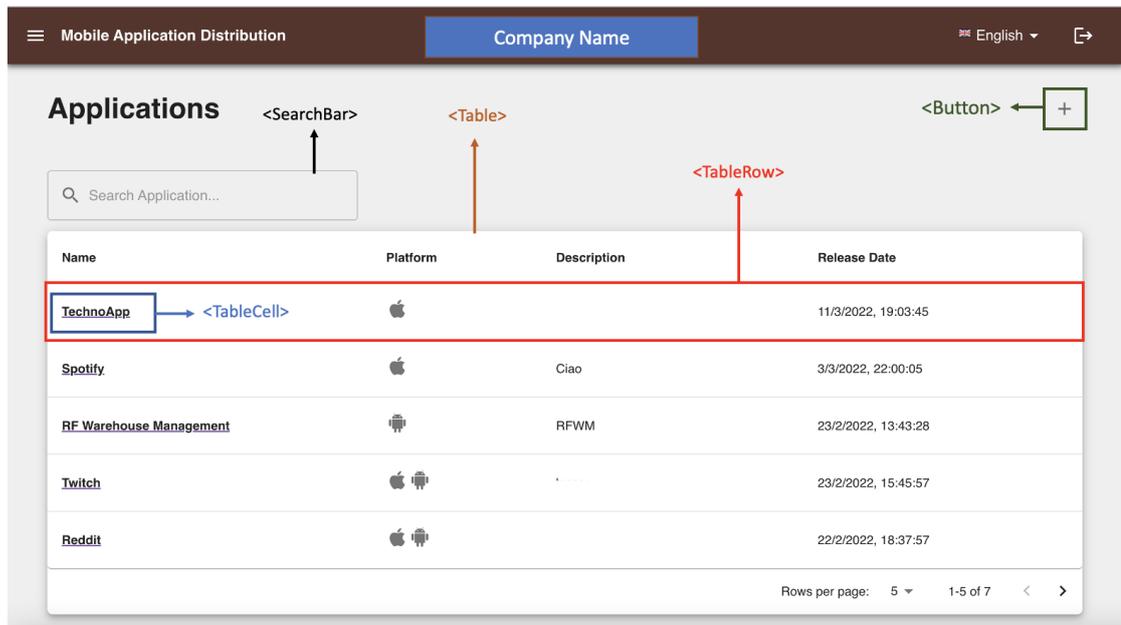


Figure 4.4: Applications Page

The name, the icon, the platform and a short description are reported for each `<TableRow>` in the `<Table>`. The possibility to look for an application by name was provided too, through a Search Box. A `<Button>` opening a `<Dialog>` for Uploading an Application was also included, further described in section 4.1.4. The `<TableCell>` containing the name is a `<Link>` React-Router component that allows to navigate from the current page to the *Application Details Page*.

4.1.4 Application Details Page

The *Application Details Page* has a very complex UI, combining many React components and information coming from the Back-end. Figure 4.5 shows the top-half portion of the designed page.

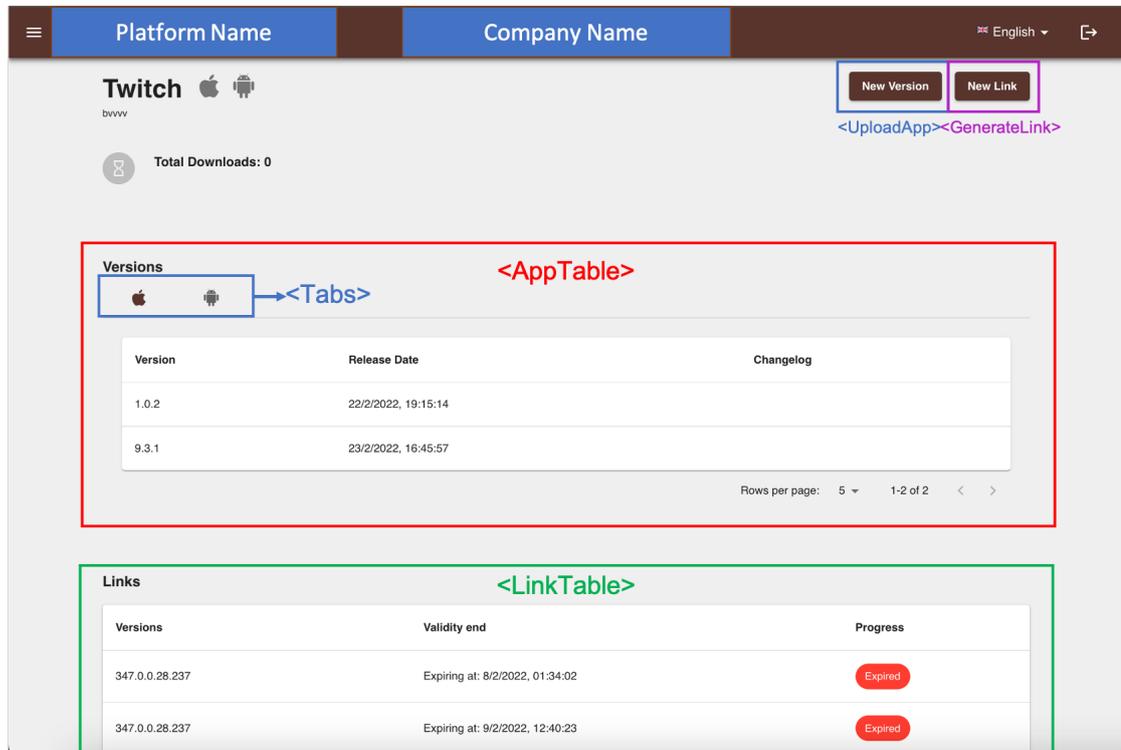


Figure 4.5: Applications Details Page: top-half page

First of all, some general information related to each application are shown: last update, platform and total amount of downloads.

Then, different versions of the same application are listed. Depending on the nature of the application (cross-platform or not), the versions are shown through two `<Tab>` allowing to navigate between different OS, or in a single table. The app chosen to be represented in Figure 4.5 is cross-platform and two `<Tab>`s are rendered.

To determine if an application is cross-platform or not, a reusable `checkPlatform` function was designed.

Listing 4.4: `checkPlatform` function

```

1  function checkPlatform(versions: Version[]) {
2      let isIos = false;
3      let isAndroid = false;

```

```

4
5     versions.filter(
6       (item: Version) => item.platform === "IOS"
7     ).length > 0 ? isIos = true : isIos = false;
8
9     versions.filter(
10      (item: Version) => item.platform === "ANDROID"
11    ).length > 0 ? isAndroid = true : isAndroid = false;
12
13    if (isIos && isAndroid) {
14      return ("Android; iOS;")
15    } else if( !isIos && isAndroid){
16      return "Android"
17    }
18    else if( !isAndroid && isIos){
19      return "iOS"
20    }
21  }

```

The `checkPlatform` function takes an array of versions related to a single application as a parameter and look at `platform` fields of each single entry. Then, it counts the occurrences of "IOS" and "ANDROID" and, based on the result of the calculation, returns a string describing the nature of the application. That string is used in TSX to render either a single `<Table>` or two tables.

Scrolling down, a Link `<Table>` containing the records of all the generated links appears. For each `<TableRow>`, a `<TableCell>` containing the related versions, the validity time and a `<LinkProgress>` component, already introduced in section 4.1.2 are shown. The possibility to reuse the same components, already built for other pages, makes coding in React very modular and clean.

The bottom part of the page reuses the `<MapChart>` and `<DownloadBarChart>` components, already introduced in section 4.1.2 and further described in section 4.6. The involved data change with respect to the Dashboard, since in this case the amount of the download is related to that particular application only.

The *Application Details* Page allows the user to load a new version of the application or to generate the link to spread the `.ipa` or `.apk` files with customers and partners. These actions are performed through two pop-ups, properly designed and accessible with two `<Button>` components rendered on top of the page.

The following listing shows how the components were nested in order to compose such a complex User Interface.

Listing 4.5: `<ApplicationDetailsPage>`

```

1  function ApplicationDetailsPage () {
2    ...
3    <UploadApp
4      buttonText={t("New Version")}

```

```

5     dialogTitle={t("Load a New Version")}
6     expectedAppID={myID}
7   />
8   <GenerateLink name={appName} versions={versions} platform={
platform}/>
9   ...
10  {platform == "Android; iOS;" ? (
11    ...
12    <TabPanel value="1">
13      <ApplicationsDetailsTable
14        columns={columns}
15        versions={versions.filter(
16          (item: Version) => item.platform == "IOS"
17        )}
18      />
19    </TabPanel>
20    <TabPanel value="2">
21      <ApplicationsDetailsTable
22        columns={columns}
23        versions={versions.filter(
24          (item: Version) => item.platform == "ANDROID"
25        )} />
26    </TabPanel>) :
27    (<ApplicationsDetailsTable columns={columns} versions={
versions} />)}
28    ...
29    <LinkDetailsTable columns={linkColumns} links={links} />
30    <MapChart />
31    <GlobalLineChart />
32    ...
33  }

```

<Stack> and <Grid> components, allowing to make the layout responsive, were omitted for sack of interest. The most important elements are reported. At lines 3 and 8 one can see how a component can be reused, by changing the particular data to be shown as a parameter. At line 10, the *conditional rendering* can be clearly visible: based on the result of `checkPlatform`, one or two <ApplicationDetailsTable> are rendered.

Upload App

The <UploadApp> component is a Material-UI <Dialog> allowing to load a new application or a new version of a pre-existing application. To be reusable, it takes two strings, representing the text on the <Button> opening the pop-up and the <Dialog> title, as parameters. It takes a third optional parameter too, the AppID which is present only on the <ApplicationDetails> Page.

The `<Dialog>` is organized by steps, using the `<Stepper>` Material-UI component. The `<Stepper>` helps providing a more interactive User Experience (UX), asking the user to fill a form in a sequential way.

The *conditional rendering* of React was again a key point for the design of `<UploadApp>`. Based on the `<Step>` active at a certain moment, different components are rendered. Figure 4.6 shows `<Step>` 1, containing an `<Input>` HTML element, to select a file from the computer.

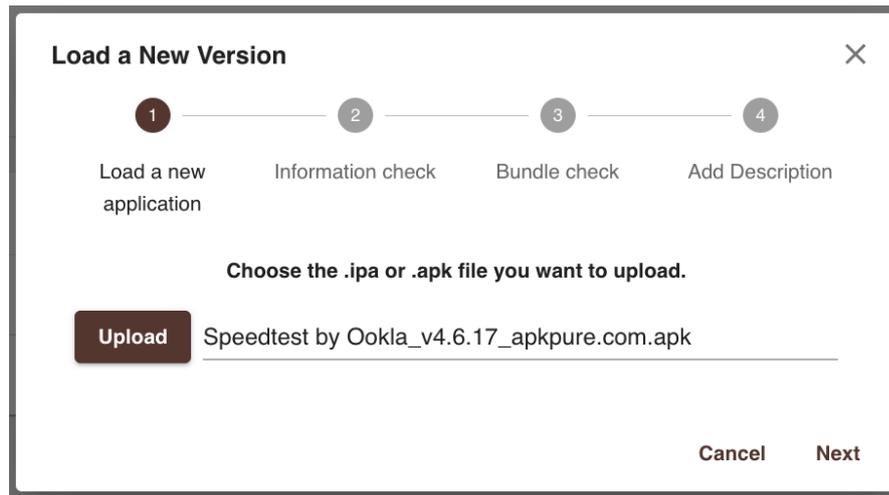


Figure 4.6: Step 1

A validation on the format is performed, by setting the `accept` parameter to `".ipa, .apk"`, so that the user cannot select files with different extension. Only if one file is selected, the user can proceed to `<Step>` 2, otherwise an `<Alert>` message is shown.

`<Step>` 2, represented in Figure 4.7, shows a `<Stack>` of `<TextField>`s automatically filled with the most important information related to the application/version: the name, the version number, the unique identifier (Bundle Identifier for iOS or Package Name for Android) and the platform. This information is returned under a JSON format from a specific designed function applied to the selected file, thanks to the external `app-info-parser` JS library [59], able to parse `.ipa` and `.apk` files. The following lines of code show the implemented parsing function.

Listing 4.6: File Parsing Function

```

1 import AppInfoParser from "app-info-parser";
2 export default async function fileSelect(file: File) {
3   const values = {
4     appName: "", version: "", packageName: "", icon: "", platform: ""
5   };

```

```

6
7  const parser = new AppInfoParser(file);
8  const result = await parser.parse();
9  if (file.name.includes(".ipa")) {
10     values.appName = result.CFBundleName;
11     values.version = result.CFBundleShortVersionString;
12     values.packageName = result.CFBundleIdentifier;
13     values.icon = result.icon;
14     values.platform = "IOS";
15 } else if (file.name.includes(".apk")) {
16     values.appName = result.application.label;
17     values.version = result.versionName;
18     values.packageName = result.package;
19     values.icon = result.icon;
20     values.platform = "ANDROID";
21 }
22 return values;
23 }

```

The designed function takes a file as an input and at line 8 it returns a JSON object containing all the information. Then, only the fields of interest are saved and returned to the parent component. The parent `<UploadApp>` component keeps these values in its state, using the React `useState` Hook.

Figure 4.7: Step 2

The user can check the information and proceed to `<Step>` 3, which is an optional step added to manage the particular case in which a mono-platform application may

become a cross-platform one. For example, an Android version of an application that until that moment was iOS-designed only, has to be loaded, or viceversa. So, only for those particular cases, the user can select the application he wants to load the version to, from a list of already present applications. Step 3 is shown in Figure 4.8.

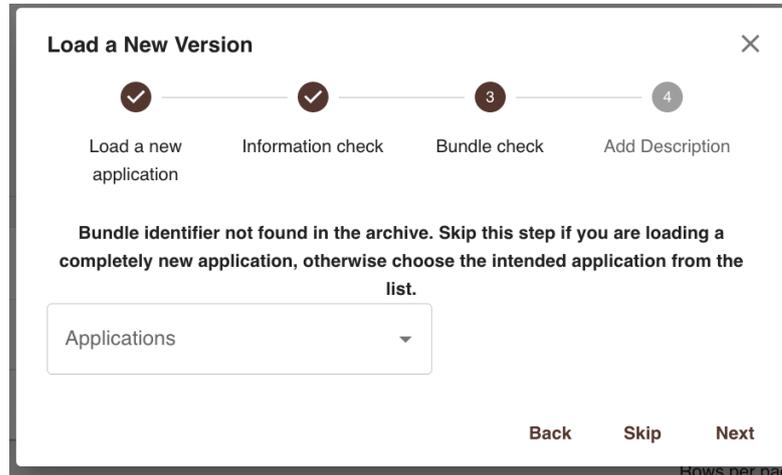


Figure 4.8: Step 3

The authenticated user can skip this step and navigate to <Step> 4, represented in Figure 4.9, where a short description of the application/version is required to be inserted.

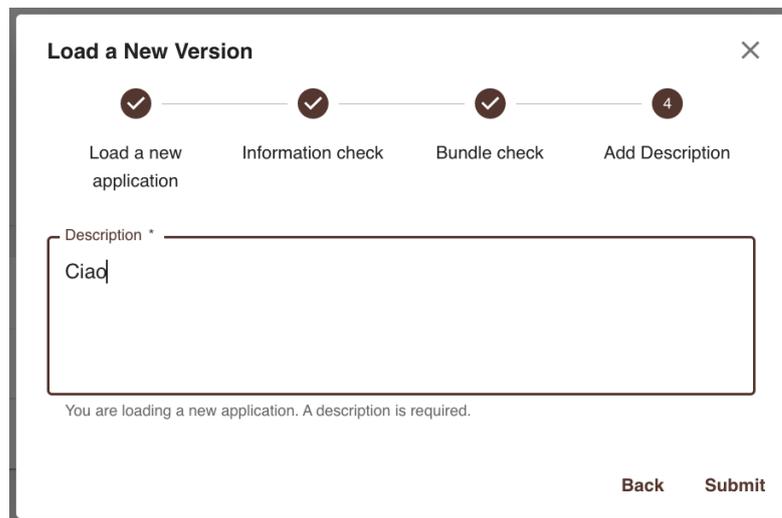


Figure 4.9: Step 4

At this point, the form can be submitted and the collected data sent to the Back-end. A `POST` request is performed to the Back-end. The body of the request contains the loaded file and the related info (name, version number, platform...) got from the parser and saved to proper variables in the State. If the user is loading a completely new application, the `POST` request will be performed in two steps. First, an application entry is created through a `POST` to the "apps" endpoint. The server sends back a unique `AppID` identifier as a response. After that, a second `POST` request is performed to the "version" endpoint and an entry is added to the `Version` table. The details of requests/responses exchanged between the Front-end and Back-end will be reported in section 4.3.

Link Generation

The `<GenerateLink>` component is a Material-UI `<Dialog>`, allowing the authenticated user to generate a link for sharing a certain application during a limited period of time. As for `<UploadApp>` described in the previous subsection, the `<Dialog>` is organized in steps, using the `<Stepper>` Material-UI component. For mono-platform applications, at most one version to be shared can be selected at a time. For cross-platform applications, at most one version per platform can be shared at a time, that is two versions belonging to different platforms can be spread using the same generated link.

`<Step>` 1, represented in Figure 4.10, contains one or two Material-UI `<Select>`, a dropdown menu to select one or two versions to be shared.

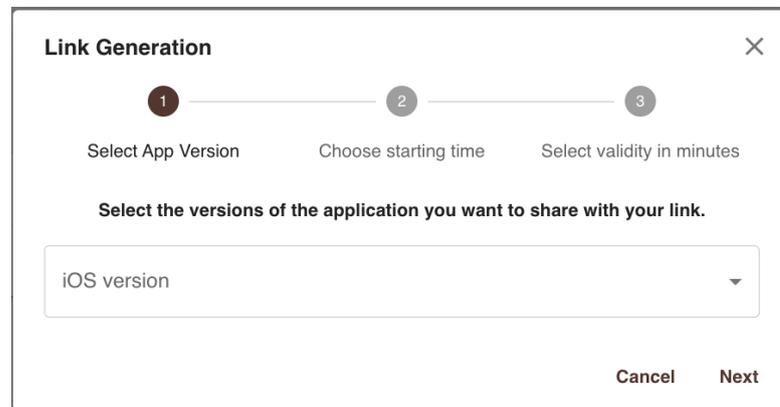


Figure 4.10: Link Generation Procedure: Step 1

If at least one version is selected, the user can navigate to `<Step>` 2, shown in Figure 4.11, where a `<DateTimePicker>` component is rendered, to choose the starting validity time of the link.

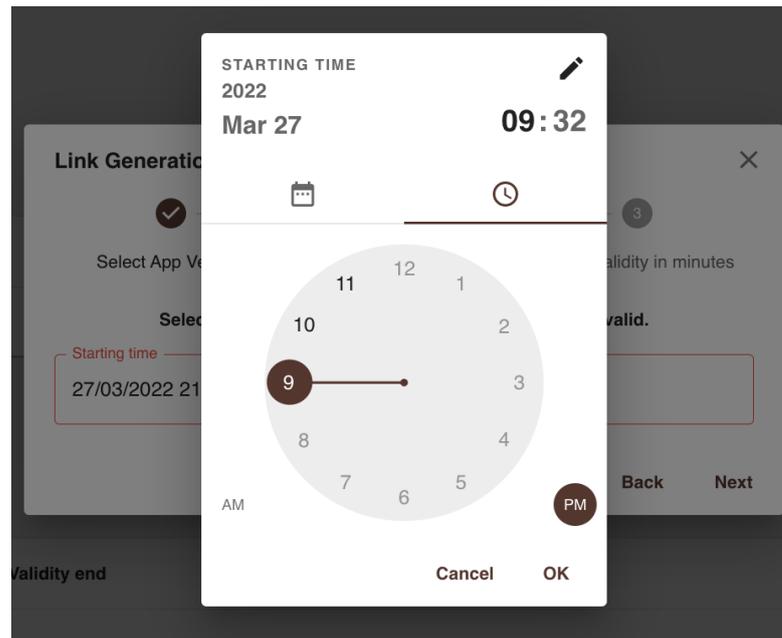


Figure 4.11: Link Generation Procedure: Step 2

<Step> 3 in Figure 4.12 shows a <TextField> to insert the validity time period of the generated link expressed in minutes. Only numbers are allowed.

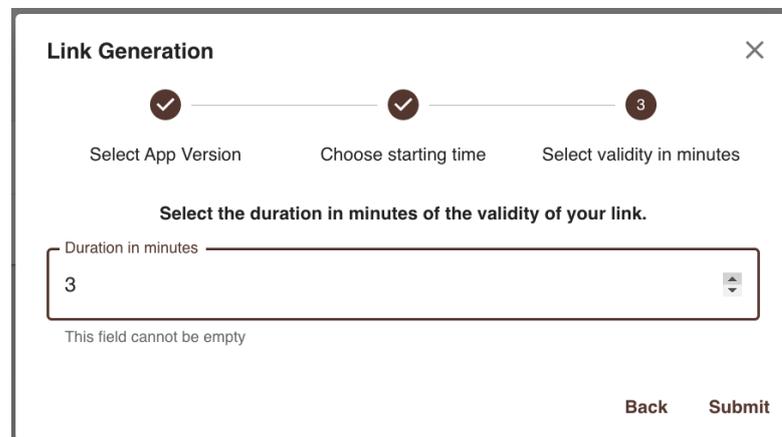


Figure 4.12: Link Generation Procedure: Step 3

At this point, the form can be submitted and the collected data sent to the Back-end. Again, the details of requests/responses exchanged between the Front-end and Back-end will be reported in section 4.3. By clicking on **Submit** button, the user is redirected to a *Download Page*, whose URL is the public link to be

shared to customers and partners to download the files. The *Download Page* is further described in section 4.1.5.

4.1.5 Download Page

The *Download Page* is the core page of the platform, allowing to download `.apk` and `.ipa` files. The main difference with respect to the previous introduced pages is its *public* nature: the final users can access it through the public link generated by one authenticated user and click on the proper Download <Button> to download/install the application of interest.

The *conditional rendering* feature of React was applied again. This time, nested conditions were taken into account. First, a check on the validity time period of the link is performed. If the link is expired or it is still not valid, a <LinkNotValid> component containing the expiration/validity start date is shown. The following lines of code shows how the components are rendered based on the result of the `compareTime` function which takes as input the `validity_start` and `validity_end` of a link and compares it with the current time `now`. If `now` is included in the time interval between `validity_start` and `validity_end`, the link is marked as `valid`. Otherwise, it is marked as `not valid` and the <LinkNotValid> component is rendered.

Listing 4.7: Conditional rendering on link validity

```

1 function DownloadPage() {
2   return (
3     {compareTime(link.validity_end, link.validity_start) === "valid" ?
4     (...):
5     <LinkNotValid />});
6 }

```

If the link is `valid`, different elements on the page are rendered depending on the Operating System of the device opening that particular link. A proper JS function able to detect the OS was implemented, reported in Listing 4.8.

Listing 4.8: Operating System Detection

```

1 function getMobileOS() {
2   const ua = navigator.userAgent
3   if (/android/i.test(ua)) {
4     return "Android"
5   }
6   else if (/iPad|iPhone|iPod/.test(ua)
7     || (navigator.platform === 'MacIntel' && navigator.
8     maxTouchPoints > 1)){
9     return "iOS"
10  }
11  return "Other"

```

 11 | }

The string containing the OS, returned by the previous function, is used to render different elements on the page. Let's consider one OS at a time, starting from "Other" which refers to Desktop Operating Systems. The Desktop Download Page is represented in Figure 4.13.

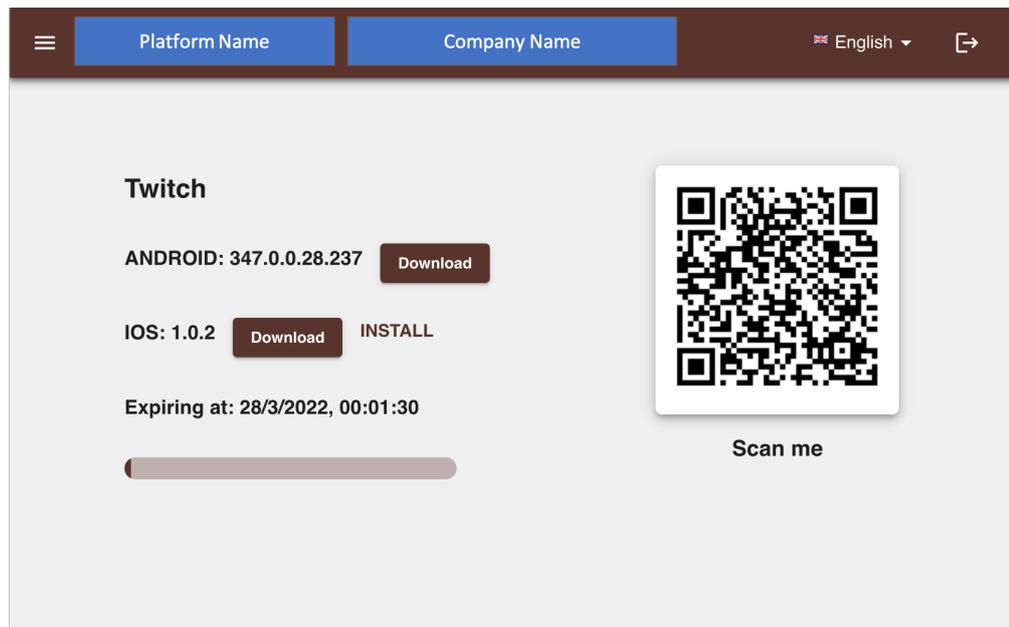


Figure 4.13: Download Page - Desktop OS

When dealing with cross-platform applications and if one version per platform is selected, both iOS and Android version numbers followed by the proper Download `<Button>`s are rendered. The expiration date and the reused `<LinkProgress>` component are reported too. The desktop version of the page also contains a QRCode, to be scanned by a mobile device to directly reach this page.

Considering Android or iOS mobile devices instead, even when dealing with cross-platform applications, only the version number relative to that particular platform and the related Download `<Button>` are rendered, when connecting to the same public URL. The real download/installation procedure will be described in Section 4.4.

4.1.6 Profile Page

The *Profile Page* is the page responsible of managing the authenticated users. Authenticated users can play 3 different roles and different components are rendered

on the page, based on their conditions.

Users can be:

- **USER:** a user able to upload the applications to the platform and to generate the links to share its own `.apk` and `.ipa` files worldwide. A normal user is enabled to use the platform by a superuser or admin.
- **SUPERUSER:** a user with higher powers with respect to the normal `user`. The superuser enables users to use the platform and assigns a role to each of them. Furthermore it also assigns a user to one or more work groups.
- **ADMIN:** a *super-partes* user, responsible of manage and monitor the platform.

The components relative to a **SUPERUSER**-oriented page are now described, since more functionalities and a more complex UI are contained. The main structure, visible in Figure 4.14, is similar to *ApplicationDetails* page one.

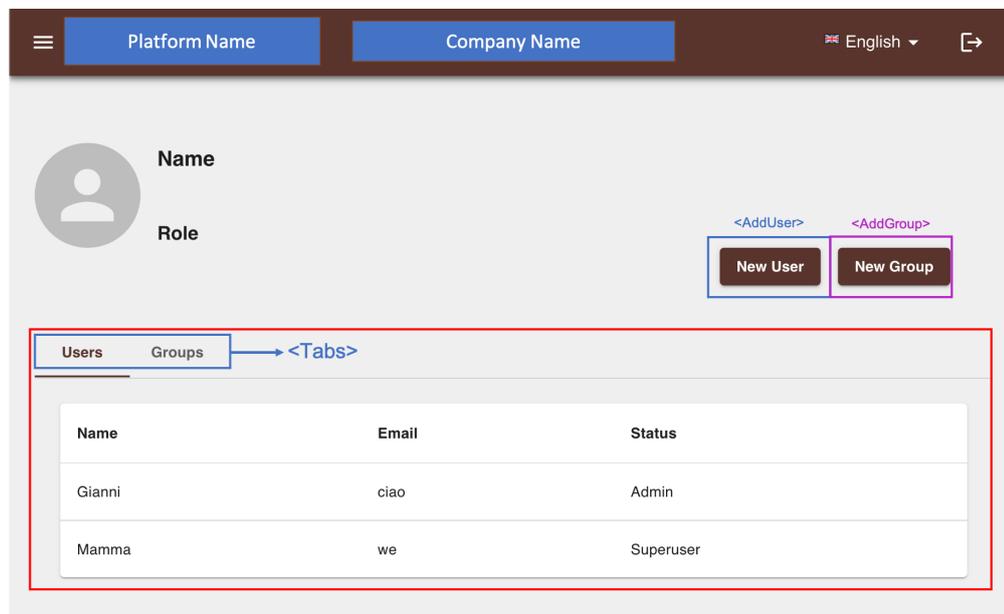


Figure 4.14: Profile Page

First of all, the name, surname and the role of the user are reported on the top-left corner.

Then two `<Tab>`s containing `Users` and `Groups` `<Table>`s are rendered. The `Users` tab shows all the active users belonging to the work groups the superuser supervises. For each `<TableRow>` in the `<Table>`, a `<TableCell>` with the name, the email that uniquely identifies the user in the organization, the role and the number of applications that user uploaded, are reported.

The `Groups` `<Table>` shows all the data related to the work groups involved. A `<TableCell>` with the name of the group, the amount of members inside the group and the number of applications coded by that group are reported.

Similarly to the pop-ups already described in section 4.1.4, two `<Dialog>`s for adding one user and one group were designed.

User creation

The `<addUser>` component is a Material-UI `<Dialog>` allowing to create one user. To make the UI coherent with the previous section, the `<addUser>` `<Dialog>` is again organized in steps, using the `<Stepper>` Material-UI component. The `<Stepper>` helps providing a more interactive User Experience (UX), asking the user to fill the form in a sequential way.

`<Step>` 1, shown in Figure 4.15, contains a `<Select>` Material-UI component, allowing to choose one element from a list of *unactive* users. Each element in the `<Select>` is characterized by its name and its unique identifier (its organization email address).

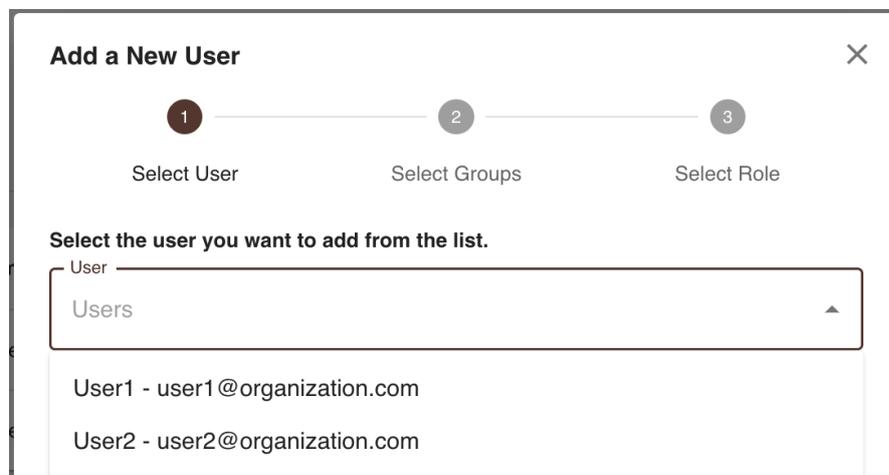


Figure 4.15: User instance creation - Step 1

It at least one user is selected from the list, one can proceed to the `<Step>` 2, where another `<Select>` Material-UI component, allowing to select one or more groups, is rendered. Step 2 is shown in Figure 4.16.

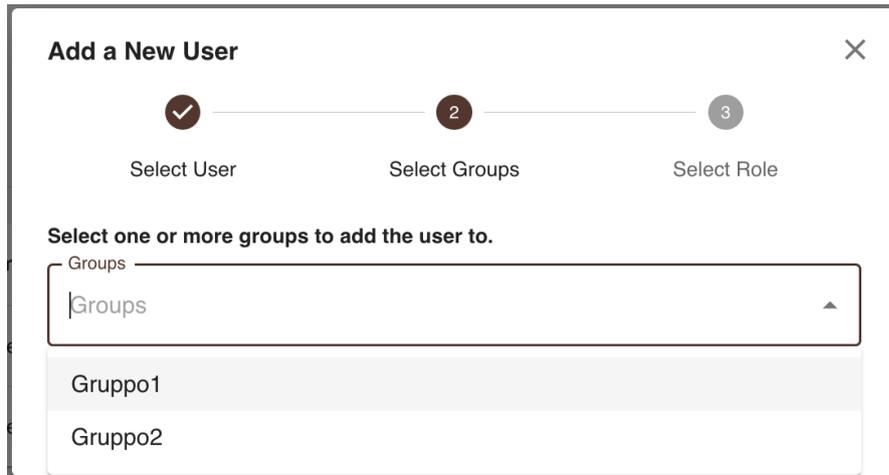


Figure 4.16: User instance creation - Step 2

When at least one group is selected, the superuser can go on to <Step> 3, represented in Figure 4.17, where a third <Select> allows to choose the role of the user to be created will play.

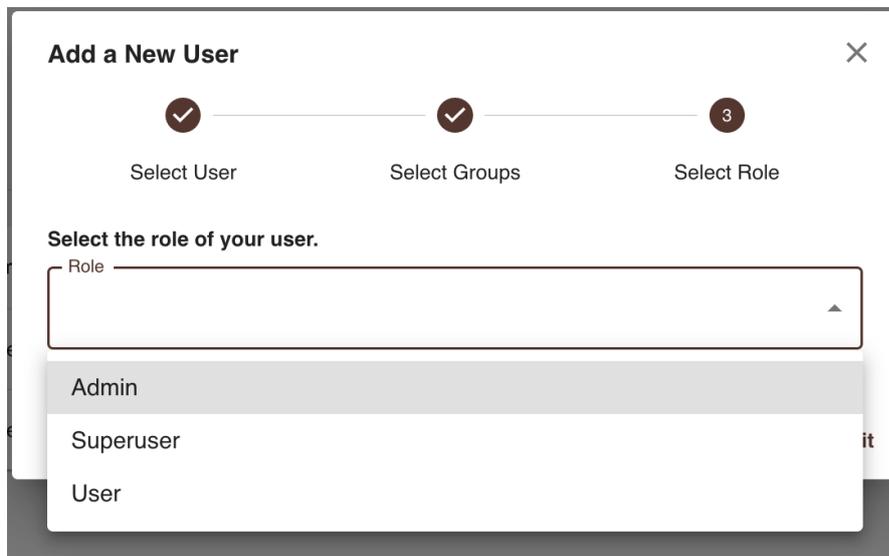


Figure 4.17: User instance creation - Step 3

At this point an entry in the `User` table in the database is created, by clicking on the `Submit` button and performing a `POST` request.

Group creation

The `<addGroup>` component is a Material-UI `<Dialog>` allowing to create one work group, containing different roled users working together. To make the UI coherent with the previous section, the `<addUser>` `<Dialog>` is again organized in steps, using the `<Stepper>` Material-UI component.

`<Step>` 1, represented in Figure 4.18, contains a `<TextField>` Material-UI component, allowing to choose the name for the group.

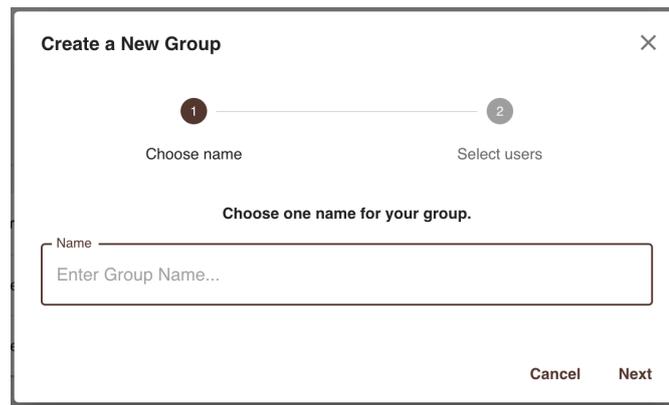


Figure 4.18: Group creation - Step 1

If the field is not empty, one can proceed to the `<Step>` 2 in Figure 4.19 where a `<Select>` Material-UI component, containing the list of all the active users, is rendered. The superuser can select one or more users and, clicking on the `Submit` button, an instance in the `Group <Table>` is added, performing a POST request.

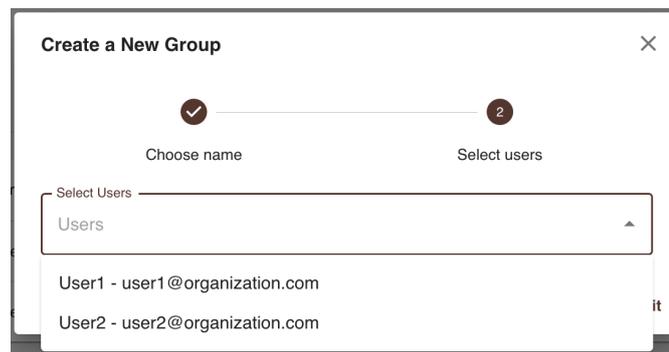


Figure 4.19: Group creation - Step 2

4.1.7 Multilanguage Support

Different languages are supported by the system. At the moment, only English and Italian are available but the amount of languages available can grow, as the client company requires it. The Multilanguage support is provided by `react-18next` library, described in 2.2.8. First of all, a configuration file with the supported languages has to be created. The system default language has to be set and saved to JS `localStorage`. `localStorage` allows JS apps to save key-value pairs in a web browser, with no expiration date. Then, one JSON file per supported language has to be created. A list of strings pairs, in which the former element is a string in the default language (English) and the latter is the translation to a certain language to be further show in the page, have to be reported in each of this JSON files. One single JSON is considered when the corresponding language acronym is selected. To change the language, the user has to click on the proper language dropdown menu in the top-right corner of the Application bar, shown in Figure 4.20.

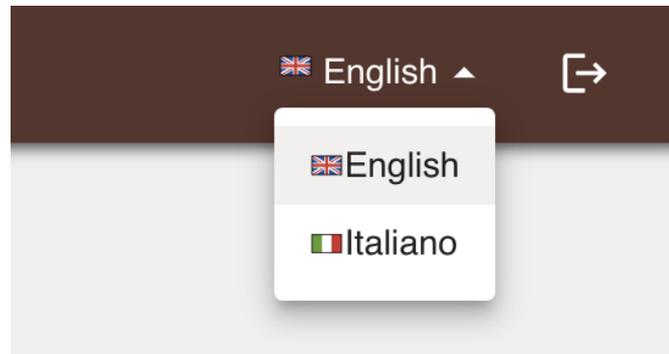


Figure 4.20: Language menu selection in Application Bar

As one can see at line 18 in 4.9, everytime one element in the menu is clicked, the `handleChange` function is triggered. This function allows to set the value of the React state variable `language` to "en" or "it" and to change the global language of the platform. All the strings present in the UI are translated to the desired language, being picked from the proper JSON file properly prepared. That language is also saved in the `localStorage` so that the user has not to change it everytime the web application is opened.

Listing 4.9: MultiLanguage Support

```

1 export default function LanguageSelect() {
2   const [language, setLanguage] = React.useState(i18next.language);
3
4   const handleChange = (event: SelectChangeEvent) => {
5     setLanguage(event.target.value as string);
6     changeLanguage(event.target.value as string);

```

```

7   };
8
9   const changeLanguage = (ln: string) => {
10      localStorage.setItem("lng", ln);
11      i18n.changeLanguage(ln);
12   }
13   return (
14     <Select
15       ...
16       value={language}
17       label={language}
18       onChange={handleChange}
19       inputProps={{
20         classes: {
21           icon: classes.icon
22         }}
23     >
24     <MenuItem value={"en"}> <Icon icon="openmoji:flag-united-
kingdom" /> English </MenuItem>
25     <MenuItem value={"it"}> <Icon icon="openmoji:flag-italy"
spacing={1}/> Italiano </MenuItem>
26     </Select>
27   );
28 }

```

To make a string translatable, an instance of the `useTranslation` Hook has to be declared in a page. After, that created instance has to be used as a wrapper for all the strings to be translated.

4.2 WebServer and Database development

This section is about the Back-end development phase, occurred in parallel with Front-end development. The Back-end development consists of the creation of the WebServer and the related Database to collect the data from. The business logic of the application is included too. Next sections will present a quick overview of the steps followed to perform these tasks, taking into account the fact that the candidate was not directly involved in these phases.

4.2.1 WebServer

First, `Node.js` has to be downloaded and installed in order to create a WebServer. `Node.js` allows developers to use JavaScript to write Back-end code, even though traditionally it was used to write Front-end [60].

`NestJS` library, introduced in section 2.3.1, was used as a framework to build a Node server-side. The command `nest new project-name` was typed in a terminal

window in order to create a new NestJS project. The `project-name` directory, containing several core files, was created. The piece of code in 4.10, extracted from `main.ts` file, shows the creation of the server-side application, through the `NestFactory` class.

Listing 4.10: `main.ts`

```

1 async function bootstrap() {
2   const app = await NestFactory.create(AppModule, {
3     logger: ['error', 'warn', 'debug'],
4     bodyParser: false
5   });
6   ...
7   const configService = app.get(ConfigService);
8   const env = configService.get('NODE_ENV');
9   const port = configService.get('PORT');
10  await app.listen(port);
11  console.log(
12    `#### app ${process.env.APP_NAME} version:${process.env.
13     APP_VERSION} started @ http://localhost:${port} with .env ${env}`,
14  );
15 }

```

At line 10, the port which the application was listening to was set. It depends on the environment the developer is using (development or production). Among the other files in `project-name` directory, `app.module.ts` is reported in 4.11. All the controllers, providers and the other modules related to all the entities declared, have to be reported in the app `@Module` instance. The integration with the database, further described in section 4.2.2 was included too, at line 4.

Listing 4.11: App Module

```

1 @Module({
2   imports: [
3     ...
4     TypeOrmModule.forRootAsync({
5       useFactory: async () =>
6         Object.assign(await getConnectionOptions(), {
7           autoLoadEntities: true,
8         }),
9     }),
10    AppsModule,
11    VersionsModule,
12    LinksModule,
13    DownloadModule,
14  ],
15  controllers: [AppController],
16  providers: [AppService],
17  ...

```

18 | })

4.2.2 Database

The Database is the part of the Back-end where all the data to be retrieved are stored. Microsoft SQL Server was chosen as a Database for the project. Microsoft SQL Server is a Relational Database Management System (RBMS) developed by Microsoft, with the primary function of storing and retrieving data.

Since the full project development was carried out in TypeScript, TypeORM was chosen as Object-Relational Mapper (ORM). An ORM is a programming technique used to convert data between Relational Databases and Object-Oriented Programming languages (OOP) [46]. It allows to write queries without writing explicit SQL queries. TypeORM helps to develop any kind of application that uses databases, from small applications with a few tables to large scale enterprise applications with multiple databases [47]. TypeORM was also chosen because it works well with `Nest.js`, already cited in previous section. To combine the two, the dedicated `@nestjs/typeorm` package was used. First of all, a `ormconfig.json` configuration file containing some useful properties related to the DB, like the type, the port it is listening to, the username and the password to access the data, etc. was created. Two instances of the database were created, one for the production mode and one for the development/test mode. The lines of code [4-8], reported in 4.11, show the integration of TypeORM in the project.

Through `getConnectionOptions`, the connection options were read from the `ormconfig` file. The Modules related to all the entities involved in the DB were included too. `Application`, `Version`, `Link` and `Download` were four entities created to manage the upload/download of the applications. `User` and `Group` entities instead were created for users management phase.

TypeORM allows to create the entities mentioned above using the `@Entity` decorator. Starting from an entity, the relative table was constructed. In 4.12 the `Application` entity was created. For each desired column in the `Application` table, a `@Column` decorator was placed before and the expected type of the field was specified.

Listing 4.12: Application Entity

```

1 @Entity()
2 export class Application {
3   @PrimaryGeneratedColumn()
4   id: number;
5
6   @Column()
7   name: string;
8

```

```

9 | @Column()
10 | description: string;
11 |
12 | @Column({ type: 'varchar', length: 'max', nullable: true })
13 | icon: string;
14 |
15 | @OneToMany(() => Version, (version: Version) => version.app)
16 | versions: Version [];
17 |
18 | @CreateDateColumn()
19 | created_at: Date;
20 |
21 | @UpdateDateColumn()
22 | updated_at: Date;
23 |
24 | @DeleteDateColumn()
25 | deleted_at: Date;
26 | }

```

The `@PrimaryGeneratedColumn` decorator allows the `id` field to be automatically incremented.

The `@CreateDateColumn`, `@UpdateDateColumn` and `@DeleteDateColumn` decorators at lines 18, 21 and 24 allow those fields to be automatically filled and updated. At line 15, the relationship between `Application` and `Version` entities is established. The `@OneToMany` decorator tells that many versions correspond to a single entry in the applications table.

Similarly to before, the following lines of code show how the `Version` table was created.

Listing 4.13: Version Entity

```

1 | @Entity()
2 | export class Version {
3 |   @PrimaryGeneratedColumn()
4 |   id: string;
5 |
6 |   @ManyToOne(() => App, (app: App) => app.versions)
7 |   app: App;
8 |
9 |   @Column()
10 |   versionNumber: string;
11 |
12 |   @Column({ nullable: true })
13 |   changelogs: string;
14 |
15 |   @Column()
16 |   released_at: Date;
17 | }

```

```

18 @Column({
19   type: 'simple-enum',
20   enum: PlatformsAvailable,
21 })
22 platform: PlatformsAvailable;
23
24 @Column()
25 bundleId: string;
26
27 @Column()
28 filename: string;
29
30 @ManyToMany(() => Link, (link: Link) => link.versions)
31 links: typeof Link [];
32
33 @CreateDateColumn()
34 created_at: Date;
35
36 @UpdateDateColumn()
37 updated_at: Date;
38
39 @DeleteDateColumn()
40 deleted_at: Date;
41 }

```

As before, a `@Column` decorator is placed before each desired column. `versionNumber`, `bundleId`, `platform` are the most important fields, able to identify each version. As opposed to before, at line 6 the `@ManyToOne` decorator tells that many entry in the versions table can correspond to a single entry in the applications one. At line 30, the `@ManyToMany` decorator establish the relationship between a `Version` and a `Link`.

In 4.14, the `Link` entity was created. It is characterized by a `validity_start` date and a `validity_end` date and an array of one or more versions. The `@ManyToMany` decorator at line 12, equal to before, shows the biunivocal relationship between `Version` and `Link` entity.

Listing 4.14: Link Entity

```

1 @Entity()
2 export class Link {
3   @PrimaryGeneratedColumn('uuid')
4   uuid: string;
5
6   @Column()
7   validity_start: Date;
8
9   @Column()
10  validity_end: Date;

```

```

11 |
12 | @ManyToMany(() => Version , (version: Version) => version.links)
13 | @JoinTable()
14 | versions: Version [];
15 |
16 | @CreateDateColumn()
17 | created_at: Date;
18 |
19 | @UpdateDateColumn()
20 | updated_at: Date;
21 |
22 | @DeleteDateColumn()
23 | deleted_at: Date;
24 | }

```

The `Download` entity will be instead introduced in section 4.6.

Considering the user management part, a `User` Table and a `Group` Table were created. The `User` Table contains an unique `id` identifier for the user (its organization e-mail), the `name` of the user and the `role` of the user in the work group.

The `Group` Table instead contains an unique `id` identifier for each group, the `name` of the group, the number of members in the group and a `users` array with reference to one or more entries in the `User` Table.

4.3 API

The first phases of the project led to the Front-end and Back-end structure development. The second phase focused on the communication between the two parts. Front-end and Back-end communicate via API. An *Application Programming Interface* (API) is a set of defined rules that explain how computers or applications communicate with one another [61]. A client application (the Front-end) performs a *request*, over the HTTP protocol, to retrieve some information. The server sends a *response* back, containing the intended data under a JSON format. The APIs are the bridge between the created Front-end and the Back-end.

This section will focus on how the APIs are served from the Back-end side and used by the Front-end.

4.3.1 Front-end

The Front-end side uses `Axios` library, already introduced in section 2.2.5, to perform the requests. These requests can be a `GET` request to retrieve a record, a `POST` request to create one, a `PUT` request to update a record, and a `DELETE` request

to delete one [61]. First of all an instance of `Axios` was created. A `baseurl.ts` file, whose content is shown in 4.15 listing, was added to the root of the project.

Listing 4.15: Axios instance creation

```
1 import axios from "axios";
2
3 const instance = axios.create({
4   baseURL: process.env.REACT_APP_BACKEND_BASE_URL,
5   headers: { Authorization: `Bearer ${localStorage.getItem("Token")}`
6   },
7 });
8 export default instance;
```

The `create` method at line 3 takes the `baseURL` to make the request to as a parameter, so that it can be omitted in the body of future requests. An `Authorization` parameter with a unique token present in the header of each request, will be further explained in section 4.5. The created instance can be imported inside any `.tsx` component that needs to `GET` data from the database or `POST` some data. As an example, the Dashboard page described in section 4.1.2 contains a huge variety of data, coming from different API endpoints. When the authenticated user loads its dashboard, several requests are made in parallel to the server.

The combined usage of `Axios` and React `useState` and React `useEffect` Hooks was needed to perform the requests. React Hooks are functions that make the developer use state and other React features without writing a specific class, but inside function components. The `useState` Hook adds React state to function components. The `useEffect` Hook instead tells React that a component needs to do something after render and make some "side effects" to be performed.

`Axios` was used to directly fetch or post the data. The following lines of code show how the Dashboard fetches the applications to be rendered in the top-left `<Card>`.

Listing 4.16: GET Apps

```
1 import api from "./base-url";
2 const [apps, setApps] = useState<Application []>([]);
3
4 useEffect(() => {
5   const fetchApps = async () => {
6     try {
7       const response = await api.get("apps");
8       setApps(response.data);
9     } catch (err: any) {
10      if (err.response) {
11        console.log(err.response.data);
12        console.log(err.response.status);
```

```

13         } else {
14             console.log('Error: ${err.message}');
15         }
16     }
17 };
18     fetchApps();
19 }, []);

```

The `apps` state variable was initialized as an empty array of `Application` type, which is an ad-hoc type whose fields correspond to the ones included in the JSON received as a response from the server. The application type contains `id`, `icon`, `name`, `versions` and `description` fields, needed to populate the UI.

After the component is rendered, the `fetchApps` function is called. At line 7, `api.get` is used inside the body of the function to `GET` the intended data. `api` was again imported from the `base-url.ts` file, so that the previous created Axios instance could be reused, without specifying the `baseURL` parameter again. Through `fetchApps` data are fetched from the server under a JSON format and saved into the React state variable `apps`, to be passed as a parameter to the `<LastApps>` component. Other data are fetched in the same way, changing the endpoint of the `GET` API.

Regarding the `POST` requests, they are performed after a form submission. Every time a component previously described in section 4.1 contains a `Submit` button, a `POST` request is involved. The piece of code in 4.17 shows a `POST` request example, referred in particular to one user.

Listing 4.17: POST Example

```

1 import api from "../base-url";
2
3 export default async function createUser(id: string, name: string,
4     status: string, groups: number[]) {
5     try {
6         let payload = { id: id, name: name, status: status, groups:
7             groups };
8         let res = await api.post("/users", payload);
9         console.log(res.status);
10    } catch (err: any) {
11        if (err.response) {
12            console.log(err.response.data);
13            console.log(err.response.status);
14        } else {
15            console.log('Error: ${err.message}');
16        }
17    }
18 }

```

Again `api` was imported from the `base-url.ts` file, so that the previous created Axios instance could be reused. The `api.post` method is called at line 7 and the API endpoint, along with a `payload` containing the data to be submitted, is set as parameters.

4.3.2 Back-end

NestJS, already described in section 2.3.1, uses *Controller* classes to manage requests. To implement the APIs, a Controller class was created for each of the entities introduced in section 4.2.2. The following lines of code show the content of the controller implemented to manage the requests related to the application entity.

Listing 4.18: Apps controller

```
1 import {
2   Controller, Get, Post, Body, Patch, Param, ... } from '@nestjs/
   common';
3 import { AppsService } from './apps.service';
4 import { CreateAppDto } from './dto/create-app.dto';
5 import { UpdateAppDto } from './dto/update-app.dto';
6
7 @ApiTags('apps')
8 @Controller('apps')
9 export class AppsController {
10   constructor(private readonly appsService: AppsService) {}
11
12   @Post()
13   @UsePipes(new ValidationPipe({ transform: true }))
14   create(@Body() createAppDto: CreateAppDto) {
15     return this.appsService.create(createAppDto);
16   }
17
18   @Get()
19   findAll() {
20     return this.appsService.findAll();
21   }
22
23   @Get(':id')
24   findOne(
25     @Param(
26       'id')
27     id: number,
28   ) {
29     return this.appsService.findOne(+id);
30   }
31   ...
32 }
```

Using the decorator `@ApiTags` at line 7, the endpoint of the API was chosen. Furthermore, for each of the CRUD methods (GET, POST, PUT, DELETE), a function was instantiated. Each of these functions contains its related decorator, as visible at lines 12,18,23 (`@Get`, `@Post`). These methods exploit other functions defined in the `apps.service` class, that is the *Service* related to the same entity. These functions describe the actions to be performed over a certain entity. The piece of code in 4.19 shows the implementation of `findAll` and `findOne` methods, used over the `App` entity.

Listing 4.19: Apps service

```

1  async findAll(): Promise<App[]> {
2      return await this.appRepository.find({
3          relations: [ 'versions' ],
4      });
5  }
6
7  async findOne(id: number): Promise<App> {
8      try {
9          return await this.appRepository.findOneOrFail(id, {
10             relations: [ 'versions' ],
11         });
12     } catch (error) {
13         throw new HttpException('Cannot find app ${error}', 404);
14     }
15 }

```

Both methods exploit pre-built functions already provided by NestJS (`.find` and `.findOneOrFail`), to fetch all the entries or just one from the `Application` Table.

A *Controller* class and a *Service* class were created for all the other entities, so that all the APIs were implemented.

4.4 Download

This section focuses on the most important aspect of the system: the **download of the applications**. In Android system, the applications code is compiled by Android SDK tools along with any data and resource files into an `APK` (Android Package) file with an `.apk` extension. An Android Package contains the contents of an Android app that are required at runtime and it is the archive file that Android-powered devices use to *install* the app [62]. For iOS instead, the installation files have an `IPA` file extension. As `APK`, they are archive files for holding the various pieces of data that make up an iPhone or iPad application.

To download and install an application using the designed system, the final users can access the public *Download Page*, already described in section 4.1.5, opening a link generated and shared by one authenticated user. According to the Operating

System (OS) the final user is using to navigate to that page, different components are rendered. Furthermore, the installation mechanism differs depending on the OS. The differences between the two platforms are described in the next subsections.

4.4.1 Download for Android

This section focuses on the download using an Android device. The final user only needs to click on the proper Download button to get the file related to a particular version of a certain application. When the click event is triggered, the `handleDownload()` function reported in 4.20, is called. It takes as a parameter the URL to perform the GET request to and the file extension of the desired application. The `responseType: "blob"` field is necessary to impose the fact that a file-like type has to be received as a response. A `Blob` (Binary Large Object) is in fact a data type that can store binary data like multimedia objects.

Listing 4.20: Download function

```

1 function handleDownload(linkText: string, extension: string, name?:
  string) {
2   api.get(linkText, {responseType: "blob"
3   }).then((response) => {
4     const link = document.createElement("a");
5     const url = window.URL.createObjectURL(new Blob([response.data
6   ]));
7     link.href = url;
8     if (name){
9       link.setAttribute("download", name + extension);
10    }
11    else {
12      link.setAttribute("download", "application-" + name + extension)
13    };
14    document.body.appendChild(link);
15    link.click();
16  });
  }

```

The same function is triggered when dealing with iOS platform but using different parameters. To ensure that the file with the right extension is received upon button click, the proper `<Button>` is coded as in 4.21.

Listing 4.21: Download Button

```

1 const linkTextAndroid = "download/" + uuid + "/ANDROID";
2 <Button variant="contained"
3   onClick={() => handleDownload(linkTextAndroid, ".apk", appName)}>
4   {"Download Android"}
5 </Button>

```

The `linkTextAndroid` URL parameter to make the request to is composed of the unique identifier (`uuid`) of the generated link, followed by a string with the right platform. The file extension is `".apk"` and the optional parameter `appName` is taken from the parameters of the URL of the page.

Once the user clicks on the `<Button>`, the download immediately starts and the user is asked to either install the application or just store the `.apk` file over the device. Figure 4.21 shows the steps followed by an Android device, in order to download an application.

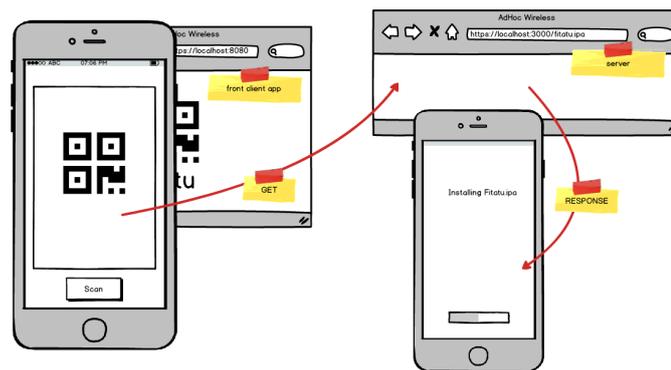


Figure 4.21: Download Procedure on Android [63]

4.4.2 Download for iOS

This section focuses instead on the download procedure on iOS. To download an `.ipa` file, it is sufficient to click on the proper download button. The same Download function cited in 4.21 is called with different parameters. The `.ipa` file is downloaded and locally stored on the device but nothing else happens. Differently from before, the user is not directly asked to install the application, but a further step has to be performed.

First of all, a so called `manifest.plist` file has to be created through the `createManifestFile` function reported in 4.22. This file contains some metadata related to the application (name, version, unique bundle identifier...) along with the URL to download the application from. This information is passed to the creation function as parameters.

Listing 4.22: Download function

```

1 export const createManifestFile = (
2   appName: string ,
3   appLink: string ,
4   appBundleId: string ,
5   appVersion: string ,

```

```

6 |   uuid ,
7 | ) => {
8 |   const manifestFile = path.join( 'manifest' , `${uuid}.manifest.plist`
9 |   );
10 |   const manifestContent = createManifestContent(
11 |     appName,
12 |     appLink ,
13 |     appBundleId ,
14 |     appVersion ,
15 |   );
16 |   return new Promise<string>((res , rej) => {
17 |     fs.writeFile( manifestFile , manifestContent , (err) => {
18 |       if (err) {
19 |         rej(err);
20 |       } else {
21 |         res( manifestFile );
22 |       }
23 |     });
24 |   });

```

As opposed to Android, a URL pointing to the produced manifest file and not directly to the .ipa file was needed. This URL is `linkTextiOS` (again composed of a unique identifier (`uuid`) of the generated link, followed by `IOS` string), with a `?installOnIOS=true` parameter set to `true` suffix. In iOS environment, URLs need a prefix containing a specific task to be performed in the system. In this phase the Apple scheme `itms-services://?action=download-manifest` pointing to the generated manifest file was used. The complete URL became a concatenation of the Apple prefix plus the `linkTextiOS` followed by `?installOnIOS=true`.

Once the user clicks on this `<Link>`, a pop-up asking the user to install the desired application will appear. If the user accepts it, a further `GET` request is performed to the link contained in the `manifest.plist` file and the application is finally downloaded. Figure 4.22 shows the installation process performed in iOS.

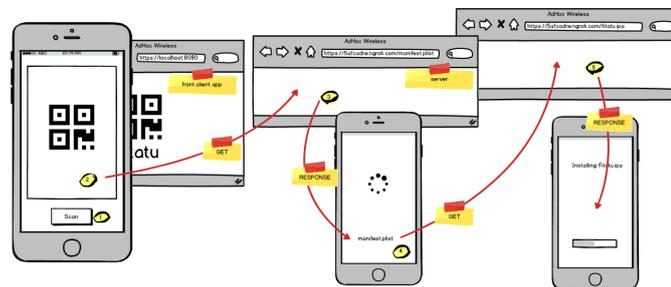


Figure 4.22: Download Procedure on iOS [63]

4.5 Microsoft Authentication Integration

Once the main architecture was set up, an authentication system is needed to allow only specific users to access the core functionalities of the platform.

This task was performed with the integration with Azure Active Directory (Azure AD), already described in section 2.4. This allows users to be authenticated using *Microsoft identity platform*, that helps to build applications in which users and customers can sign in with their Microsoft work or social accounts and provide authorized access to the designed APIs [50], without the need of designing an ad-hoc identity provider from scratch.

Two actors play a role in using the system: the users authenticated to the platform and the final users. Their differences and MSAL employment will be described in next sections.

4.5.1 Authenticated users

The authenticated users are able to login to the platform and navigate through all the pages presented in section 4.1. Furthermore, all the implemented functionalities are accessible by a user authenticated to the system only.

Considering the Front-end, Microsoft already provides `@azure/msal-react` library to authenticate users in React Single Page Applications. Before to use the library, the client company needed to register the Front-end application in Azure AD portal to get a `clientId` and a `tenantId` for configuration. The following lines of code show how to integrate MSAL and are reported in the root of the project.

Listing 4.23: MSAL React

```
1 const msalInstance = new PublicClientApplication(msalConfig);
2
3 export const msalConfig: Configuration = {
4   auth: {
5     clientId: "client-id",
6     authority:
7       "https://login.microsoftonline.com/tenant-id",
8     redirectUri: "/",
9   }
10 };
```

To use MSAL, a `PublicClientApplication` object has to be declared. It takes as a parameter a configuration file, where the provided `clientId`, `tenantId` and a `redirectUri` are declared. When a user tries to access to the application characterized by that particular `clientId`, it is redirected to the login page of the organization represented by that `tenantId`. If the user is correctly authenticated, it is redirected to the "/" route, corresponding to the Dashboard Page, as introduced in section 4.1.1

MSAL-React already provides a pre-built set of components and Hooks, making the integration with the authentication easier. First of all, `<MsalProvider>` component has to be declared at the top level of the component tree that requires access to authentication, as a wrapper for all the `<App>`. All the children components will have access to `@azure/msal-react` context, hooks and components.

The library already provides useful components as `<AuthenticatedTemplate>` and `<UnauthenticatedTemplate>`, allowing to conditionally render part of the pages or entire pages to authenticated users only. The lines of code in 4.24 show that, navigating to the same path of the application, an authenticated user can access to all the `<Pages>` wrapped inside an `<AuthenticatedTemplate>` component, while an unauthenticated user can see the element wrapped inside the `<UnauthenticatedTemplate>` only. In the last case, a `<WelcomePage>` with a proper `<SignInButton>` is rendered. By clicking on that button, a `loginRequest` to the `authority` URL provided in 4.23 at line 6 is performed and the user can authenticate to the system using its own organization account.

Listing 4.24: Authenticated/Unauthenticated

```
1 function App() {
2   return (
3     <div>
4       <AuthenticatedTemplate>
5         <Pages />
6       </AuthenticatedTemplate>
7       <UnauthenticatedTemplate>
8         <WelcomePage />
9       </UnauthenticatedTemplate>
10    </div>
11  );
12 }
```

As before, the client company needed to register the Back-end application in Azure AD portal to get a `clientId` and a `tenantId` for configuration. To integrate Azure AD, the `passport-azure-ad` library provided by Azure was implied with `NestJS` and a `BearerStrategy` was chosen. `BearerStrategy` uses Bearer Tokens protocol to protect APIs. A user sends a request with a Bearer token in the header to the Back-end. `Passport` extracts and validates the received token, and verify if the user is authorized to get the data. In case of error, an `Unauthorized` response is sent back.

The integration with the Back-end requires the extension of the `Passport Strategy` class, already provided by `NestJS`. The options related to the authentication service (`azure-ad`) and strategy (`BearerStrategy`) have to be set. To connect Back-end and Front-end applications, the Front-end `clientId` has to be passed to the constructor at line 10.

Listing 4.25: Backend Passport

```

1 export class AzureADStrategy extends PassportStrategy(
2   BearerStrategy ,
3   'azure-ad' ,
4 ) {
5   constructor(private configService: ConfigService) {
6     super({
7       identityMetadata: 'https://login.microsoftonline.com/${
8         configService.get(
9           'TENANT_ID' ,
10          )}/v2.0/.well-known/openid-configuration ' ,
11       clientID: '${configService.get('FRONTEND_CLIENT_ID')}' ,
12     });
13 }

```

When a user signs-in, he/she needs to acquire this unique Bearer token, to be set in the header of all the requests to the Back-end. The `idToken` is acquired using the `acquireTokenSilent` function, already provided by MSAL and visible in line 8 of 4.26. It is saved in JS `localStorage` and added to the Axios instance previously created and introduced in section 4.3.

Listing 4.26: Client Token

```

1 useEffect(() => {
2   function RequestAccessToken() {
3     instance
4       .acquireTokenSilent(request)
5       .then((response) => {
6         localStorage.setItem("IdToken", response.idToken);
7       })
8       .catch((e) => {
9         instance.acquireTokenPopup(request).then((response) => {
10          localStorage.setItem("IdToken", response.idToken);
11        });
12      });
13   });
14 }
15 RequestAccessToken();
16 }, [instance]);

```

If the token is successfully acquired, the user is able to perform all the requests and get the data of interest, to populate the pages.

4.5.2 Final users

The final users are not able to authenticate to the platform. They can be clients or external partners of the company, interested in receiving a particular enterprise

mobile application to work with. The platform can also be used to test some beta versions of new apps tuned by one of the authenticated user. In this case, a final user can also be a company employee not present in the users pool able to access to the system, asked to test some functionalities of a certain application.

4.6 Analytics

This section focuses on the analytical part where all the statistical analyses are performed. Some additional functionalities were added to the Front-end and Back-end already developed. These features help the authenticated users to monitor the spread of their applications, shared through the generation of public links. The pages previously described in section 4.1 were populated with charts and maps, providing insights related to the amount of downloads. Thanks to these insights, the user can monitor the trend of the downloads of its own applications and identify possible malicious behaviours, in order to preserve the secure sharing of the applications. These statistics differ from one user to another.

4.6.1 Database update

To implement the new statistical features, a new `Download` Table was added to the Database, previously described in section 4.2.2. Each entry in the `Download` Table contains the `timestamp` at which the download was done, a `uuid` field representing the particular link used to perform the download and the corresponding version of the application. Then, the public IP Address and the corresponding country for a certain request of download received by the server are saved to other two columns. The country where a request comes from can be derived from the IP Address using the *Geoip-lite* JS library [64], able to parse IP addresses and geo-locate them.

4.6.2 Front-end components

In the Front-end, `<MapChart>`, `<DownloadBarChart>` and `<DownloadLinkChart>` ad-hoc reusable React components, already cited in 4.1.2 and 4.1.4, were created to include interesting statistical features about the application downloads.

MapChart

`<MapChart>` was realized using *React Single Maps* library, quickly introduced in section 2.2.7.

`<MapChart>` is a Material-UI `<Card>` showing a `<ComposableMap>` wrapper with a `<Geographies>` component inside. `<Geographies>` takes as a parameter a geojson file previously prepared, representing the whole world without annotations. The

user can change the area of the world it is interested in, by clicking on the proper button on the top-right corner of the <Card>. This way, the projection parameters of <ComposableMaps> changes and a different region is shown. <MapChart> and the relative dropdown menu is represented in Figure 4.23.

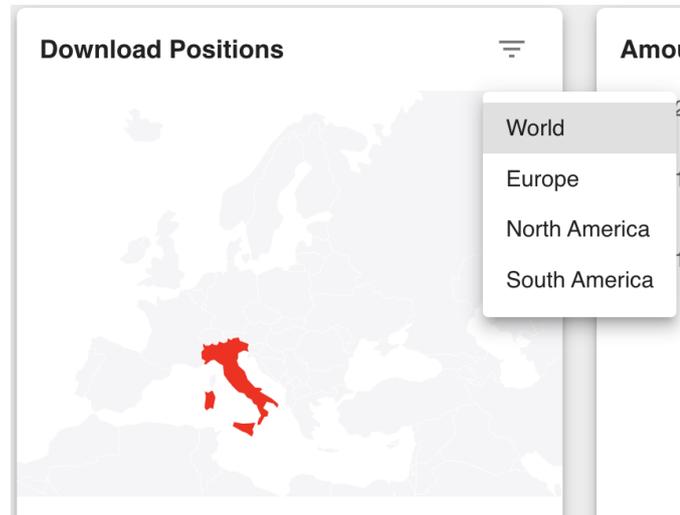


Figure 4.23: <MapChart>

This component takes as a parameter an array of **Country4Plot** type, containing a **country** and the related amount of **downloads** performed. For each entry in the array passed as *props*, a <Geography> element in the map is created and the corresponding area is colored using a **colorScale** function that scales the colors proportionally between the minimum and the maximum number of downloads in the array, creating a HeatMap.

Ad-hoc APIs were created to send to the Front-end the correct data to be shown through maps and charts. The related endpoint allowing to build this map is **statistics/countries/apps** and returns an array of **Country** type, whose entries are composed of a **country** and a corresponding array of **apps** with the related amount of downloads for that country. To build the map, a **Country4Plot** array composed of the country and the corresponding amount of downloads only is needed, so a procedure of *aggregation* of the data per country was performed.

When used in the Dashboard, <MapChart> component contains the amount of downloads per country, aggregated by *all the applications* uploaded by the user. In the Application Details Page instead, it shows the amount of downloads aggregated by *all the versions* of one single application.

The authenticated users can monitor the geographical spread of their applications looking at the designed map. If one particular application is shared with partners belonging to a certain geographical area, downloads in different areas should not

appear on the map!

DownloadBarChart

<DownloadBarChart> was created using *Recharts* library, already introduced in section 2.2.6. <DownloadBarChart>, shown in Figure 4.24, is a <BarChart> aggregating the amount of downloads per day. It shows data related to last 10 days of usage of the platform.

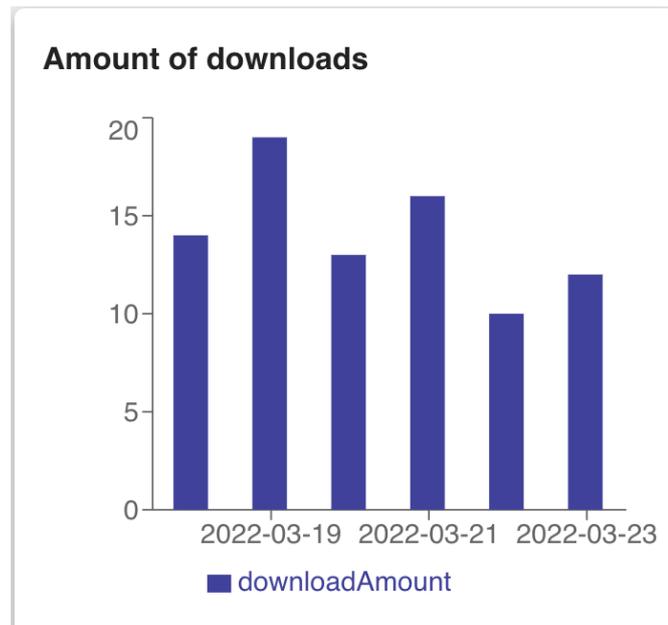


Figure 4.24: <DownloadBarChart>

Ad-hoc APIs were created to send to the Front-end the correct data to be shown through the chart. The related endpoint allowing to build this chart is `statistics/downloads/apps`, returning an array of `Version4Download` type, whose entries are composed of an `application` and a corresponding array of `downloads` which in turns contains the amount of downloads for each version of the same application.

The designed component takes an array of `Download4Plot` type, which contains a date field and the corresponding amount of downloads. To build the chart, a procedure of aggregation of the downloads per date was necessary.

When used in the Dashboard, <DownloadBarChart> component contains the total amount of downloads per date, looking at *all the applications* uploaded by the user. In the Application Details Page instead, it shows the total amount of downloads, looking at *all the versions* of one single application.

The authenticated users can monitor the average amount of downloads of their applications, looking at this chart. Possible malicious behaviours can be intercepted, when a `<Bar>` overcomes the others.

DownloadLinkChart

`<DownloadLinkChart>` was still created using *Recharts* library, introduced in section 2.2.6. It contains a `<ComposedChart>`, combining a `<BarChart>` and an `<AreaChart>`. The former accumulates the amount of downloads per generated link; the latter shows the maximum amount of downloads for the same generated links, granted during the link creation phase. `<DownloadLinkChart>` is shown in Figure 4.25.

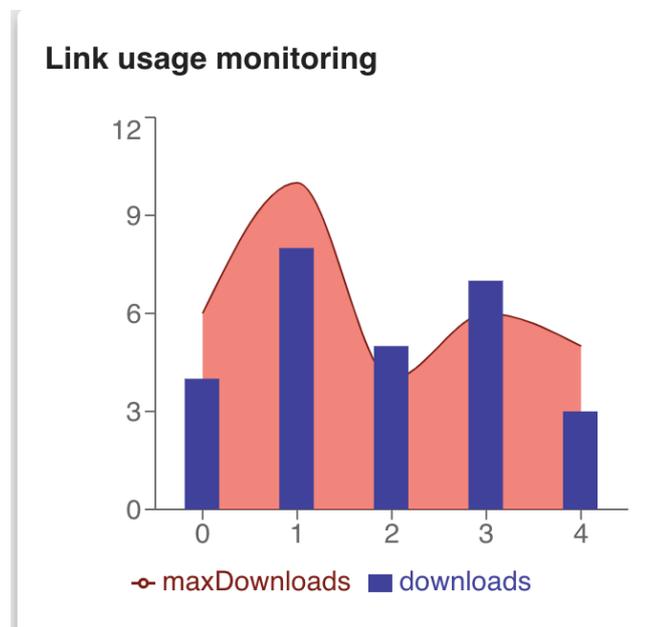


Figure 4.25: `<DownloadLinkChart>`

The authenticated users can compare the amount of downloads performed for a certain generated link and the total amount of downloads allowed for the same link. If the number of downloads overcomes the threshold set during the link generation phase (as in `<Bar>` 2 and 3 of Figure 4.25), the user can intercept possible malicious behaviours, since that link has been used more than expected.

The user can use this feature in order to allow the download of the application just to a limited portion of final users only.

4.7 Deployment on Azure Virtual Machine

All the functionalities described in previous sections were only developed on `localhost`. The employment of an Azure Virtual Machine (VM) provided by the client company was necessary, to host the designed Web Application and expose it to the Internet.

4.7.1 Deployment

Considering the Front-end, the deployment for production of the designed React application was performed by running the command `npm run build` in a terminal window. This command creates a `build` directory with a production build for the application [65]. Through that command, a React app is minified into simple HTML, CSS, and JavaScript files [66] that can be interpreted by the browser. The obtained folder serves the React application through a single entry point corresponding to `index.html` file, where the entire app resides.

After that, the `build` directory was uploaded to a shared folder belonging to the developer company drive. The developer needed to login to the Virtual Machine, open a browser window and reach the same folder just uploaded, in order to drag it from the drive and move it on the VM.

Let's now consider the Back-end. First, locally on the developer machine, the command `npm run create:zip` was run in order to create a `.zip` folder of the entire repository. Again the `.zip` directory just created was uploaded to a shared folder in the developer company drive and locally downloaded to the VM, by accessing to the same path.

Furthermore, the files in the `.zip` folder were extracted and the command `npm run azure:build:prod` was run in a terminal window. This command installed all the needed dependencies, built the application and started the server. This command creates the production instance of the Web application, served on local port 3000. Since two instances of the Web Application were needed (one for production and one for development/test), `npm run azure:build:dev` command was run in another terminal window to get the second instance devoted to the test, served on local port 3999.

Figure 4.26 and Figure 4.27 show the two commands run in two distinct terminal windows and both servers successfully started on different ports.

```
Select npm run start:prod
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\c.chieppa> cd ..
PS C:\Users> cd ..
PS C:\> cd .\OTA-Deploy\dev
PS C:\OTA-Deploy\dev> cd .\backend-latest\
PS C:\OTA-Deploy\dev\backend-latest> npm run azure:build:dev

> ota-backend@0.0.1 azure:build:dev
> npm install && npm run build && cross-env NODE_ENV=development npm run start:prod

up to date, audited 924 packages in 3s

82 packages are looking for funding
  run `npm fund` for details

5 vulnerabilities (1 moderate, 7 high)
To address all issues, run:
  npm audit fix
Run `npm audit` for details.

> ota-backend@0.0.1 prebuild
> rimraf dist

> ota-backend@0.0.1 build
> nest build

> ota-backend@0.0.1 start:prod
> node dist/src/main

### app DEVELOPMENT - Ferrero OTA version:0.1.1 started @ http://localhost:3999 with .env development
```

Figure 4.26: Development instance

```
npm run start:prod
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\c.chieppa> cd ..
PS C:\Users> cd ..
PS C:\> cd .\OTA-Deploy\
PS C:\OTA-Deploy> cd .\prod\
PS C:\OTA-Deploy\prod> cd .\release2-backend\
PS C:\OTA-Deploy\prod\release2-backend> npm run azure:build:prod

> ota-backend@0.0.1 azure:build:prod
> npm install && npm run build && cross-env NODE_ENV=production npm run start:prod

up to date, audited 991 packages in 4s

82 packages are looking for funding
  run `npm fund` for details

5 vulnerabilities (3 moderate, 5 high)
To address all issues, run:
  npm audit fix
Run `npm audit` for details.

> ota-backend@0.0.1 prebuild
> rimraf dist

> ota-backend@0.0.1 build
> nest build

> ota-backend@0.0.1 start:prod
> node dist/src/main

### app Ferrero OTA version:0.1.1 started @ http://localhost:3000 with .env production
```

Figure 4.27: Production instance

At this point, Front-end and Back-end were moved to the Virtual Machine and successfully running. Next section will focus on the setup of Internet Information Services (IIS), in order to expose the Web Application to the Internet.

4.7.2 IIS configuration

First, the installation of IIS was necessary. Inside the *Server Manager* Portal, a list of servers was presented. The server hosting the Web Application was selected from the list and **WebServer** was set as its role. Once IIS was installed, IIS Manager Tool could be opened and several operations could be performed.

Right-clicking on the server, a **new website** was created. Figure 4.28 shows the form to be filled to create a new website, using *IIS Manager Tool*.

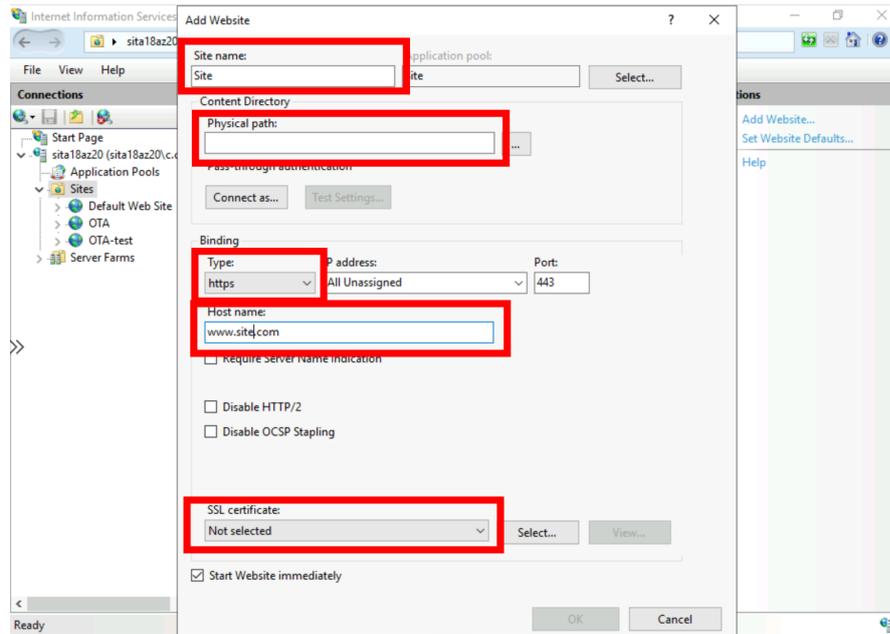


Figure 4.28: New website creation with IIS

The physical path of the Front-end build folder needed to be provided in the **Content Directory** field. The hostname and the port over which the website will be served were chosen. The protocol type (HTTP or HTTPS) had to be selected too. HTTPS uses the SSL/TLS protocol to encrypt communications. SSL/TLS confirms that a website server is who it says it is [67]. HTTPS was selected and a valid SSL certificate provided by the client company was added to the website.

Two websites were created: one for production served on port 443 and one for development/test on port 3443. Both websites use the same SSL certificate and shows the same Front-end.

The IIS configuration procedure allowed to create two separated websites, available to the Internet. A further step needed to be performed: forwarding the requests for the Back-end coming from that websites at `https://hostname:port/api/...` to `localhost:3000` and `localhost:3999` of the VM, where the production and

development instances of the Back-end were already running and ready to answer, as introduced in previous section.

To perform this task, the installation of URL Rewrite Module of IIS was necessary. This module modifies the requested URL before the Web server decides which handler to use to process the request [58].

Two new In-bound rules has to be added to convert every request. The Dev-Rule forwards the requests coming on `https://hostname:3443/api/...` to `localhost:3999`.

The Prod-Rule forwards the requests coming on `https://hostname/api/...` to `localhost:3000`. Figure 4.29 and Figure 4.30 show the configuration of the in-bound rules to forward the requests to the right server.

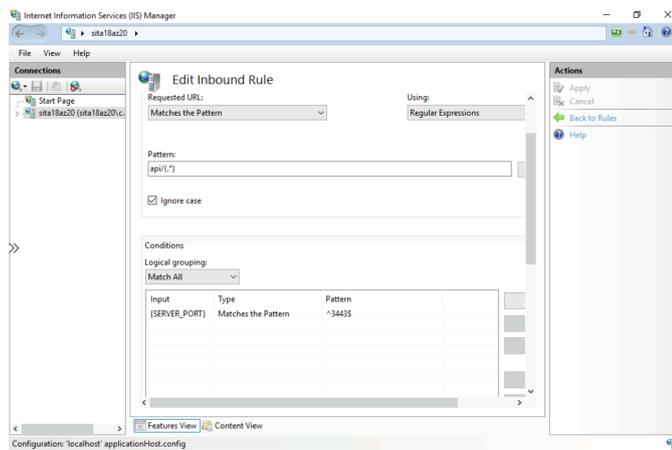


Figure 4.29: IIS - Rule Definition

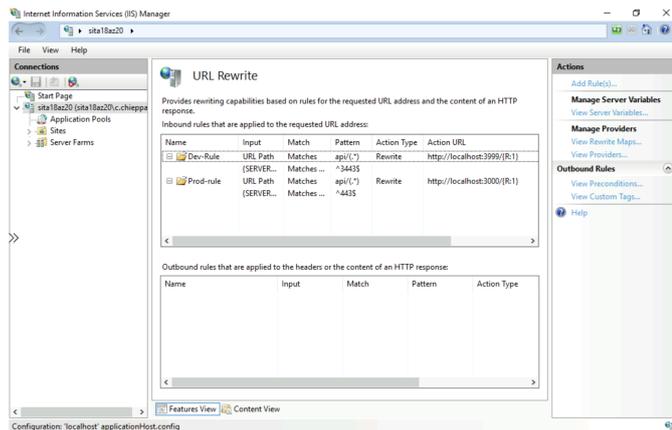


Figure 4.30: IIS - Rules Overview

Chapter 5

Conclusions

In this thesis, an enterprise distribution platform for mobile applications was developed on behalf of a leading food company. The company needed to share its own enterprise applications with customers and partners in a private, direct and secure way. Possible privacy issue can be encountered using public app stores because restricted and confidential information, contained in enterprise mobile applications, can be accessed not only by employees but also by unauthorized users. This issue was overcome designing an architecture for distributing the enterprise mobile applications over-the-air (OTA), sharing public links that are valid only for a limited amount of time, avoiding that too many unwanted downloads could be performed. Furthermore, in order to identify possible malicious behaviors, some analytical considerations were carried out to include security features by monitoring the downloads during the link validity time. The entire system architecture was designed and exposed over the Internet. A test session of all the implemented functionalities gave some first promising results. As soon as the system will be delivered to the client company, more insights about the performances will be available.

The designed system can be enhanced with the following functionalities:

1. Possibility to stop the validity of a link, when the number of the related downloads overcomes a threshold set during the link generation phase.
2. Restriction of link validity only on a certain geographical region. A final user belonging to a country outside of the selected region will see an error page instead of the public *Download* Page.
3. Automatic tests on the reliability of the applications loaded to the platform.
4. Scalability of the system on multi-servers and multi-databases, in case more instances of the same web application are required.

Bibliography

- [1] Dr. MD Rashedul Islam and Tridib Mazumder. «Mobile Application and Its Global Impact». In: (Dec. 2010) (cit. on p. 1).
- [2] URL: <https://buildfire.com/app-statistics/> (cit. on p. 1).
- [3] URL: <https://www.emergeinteractive.com/insights/detail/the-best-mobile-enterprise-application-platforms/> (cit. on p. 1).
- [4] URL: <https://www.hubspire.com/what-is-enterprise-mobile-application-development/> (cit. on p. 1).
- [5] URL: <https://definitions.uslegal.com/d/digital-distribution/> (cit. on p. 1).
- [6] URL: <https://appleinsider.com/inside/app-store> (cit. on p. 1).
- [7] URL: <https://www.apple.com/app-store/> (cit. on p. 1).
- [8] URL: <https://developer.apple.com/app-store/review/> (cit. on p. 2).
- [9] URL: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#a-typed-superset-of-javascript> (cit. on p. 4).
- [10] Christopher Nance. *TypeScript Essentials*. Oct. 2014 (cit. on p. 4).
- [11] URL: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-assertions> (cit. on p. 4).
- [12] URL: <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps> (cit. on p. 5).
- [13] URL: <https://reactjs.org/docs/components-and-props.html> (cit. on p. 5).
- [14] URL: <https://reactjs.org/docs/hooks-state.html> (cit. on p. 6).
- [15] URL: <https://reactjs.org/docs/hooks-effect.html> (cit. on p. 6).
- [16] URL: <https://reactjs.org/docs/introducing-jsx.html> (cit. on p. 6).
- [17] URL: <https://mui.com/> (cit. on p. 6).

- [18] Adam Boduch. *React Material-UI Cookbook: Build captivating user experiences using React and Material-UI*. Mar. 2019 (cit. on p. 6).
- [19] URL: <https://mui.com/components/autocomplete/> (cit. on p. 7).
- [20] URL: <https://mui.com/components/buttons/> (cit. on p. 7).
- [21] URL: <https://mui.com/components/cards/> (cit. on p. 7).
- [22] URL: <https://mui.com/components/chips/> (cit. on p. 7).
- [23] URL: <https://mui.com/components/dialogs/> (cit. on p. 7).
- [24] URL: <https://mui.com/components/progress/> (cit. on p. 7).
- [25] URL: <https://mui.com/components/selects/> (cit. on p. 7).
- [26] URL: <https://mui.com/components/stepper/> (cit. on p. 7).
- [27] URL: <https://mui.com/components/tables/> (cit. on p. 7).
- [28] URL: <https://mui.com/components/text-fields/> (cit. on p. 7).
- [29] URL: <https://reactrouter.com/docs/en/v6/api> (cit. on p. 7).
- [30] URL: <https://reactrouter.com/docs/en/v6/api#browserrouter> (cit. on p. 7).
- [31] URL: <https://reactrouter.com/docs/en/v6/api#routes-and-route> (cit. on p. 7).
- [32] URL: <https://reactrouter.com/docs/en/v6/api#usenavigate> (cit. on p. 7).
- [33] URL: <https://reactrouter.com/docs/en/v6/api#useparams> (cit. on p. 8).
- [34] URL: <https://www.digitalocean.com/community/tutorials/js-axios-vanilla-js> (cit. on p. 8).
- [35] URL: <https://www.javatpoint.com/react-axios> (cit. on p. 8).
- [36] URL: <https://github.com/recharts/recharts/blob/master/README.md> (cit. on p. 8).
- [37] URL: <https://recharts.org/en-US/api/BarChart> (cit. on p. 8).
- [38] URL: <https://www.npmjs.com/package/react-simple-maps> (cit. on p. 8).
- [39] URL: <https://www.react-simple-maps.io/docs/composable-map> (cit. on p. 8).
- [40] URL: <https://www.react-simple-maps.io/docs/geographies/> (cit. on p. 8).
- [41] URL: <https://www.react-simple-maps.io/docs/geography/> (cit. on p. 8).

- [42] URL: <https://lokalise.com/blog/how-to-internationalize-react-application-using-i18next/> (cit. on p. 8).
- [43] URL: <https://react.i18next.com/> (cit. on p. 9).
- [44] URL: <https://docs.nestjs.com/> (cit. on p. 9).
- [45] URL: <https://enlear.academy/why-you-should-use-nestjs-as-your-backend-framework-bd1ff1acce5d> (cit. on p. 9).
- [46] URL: https://www.tutorialspoint.com/hibernate/orm_overview.htm (cit. on pp. 9, 39).
- [47] URL: <https://typeorm.io/#/> (cit. on pp. 9, 39).
- [48] URL: <https://azure.microsoft.com/en-us/services/active-directory/#overview> (cit. on p. 10).
- [49] URL: <https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-what-is> (cit. on p. 10).
- [50] URL: <https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-overview> (cit. on pp. 10, 14, 50).
- [51] URL: <https://docs.microsoft.com/en-us/azure/active-directory/develop/authentication-vs-authorization> (cit. on p. 10).
- [52] URL: <https://docs.microsoft.com/en-us/azure/active-directory/develop/msal-overview> (cit. on p. 10).
- [53] URL: <https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-browser#about> (cit. on p. 10).
- [54] URL: <https://github.com/AzureAD/microsoft-authentication-library-for-js/tree/dev/lib/msal-react> (cit. on p. 10).
- [55] URL: <https://www.iis.net/> (cit. on p. 10).
- [56] URL: <https://www.techtarget.com/searchwindowsserver/definition/IIS> (cit. on p. 10).
- [57] URL: <https://docs.microsoft.com/it-it/iis/get-started/introduction-to-iis/introduction-to-iis-architecture> (cit. on p. 11).
- [58] URL: <https://docs.microsoft.com/en-us/iis/extensions/url-rewrite-module/iis-url-rewriting-and-aspnet-routing> (cit. on pp. 11, 60).
- [59] URL: <https://www.npmjs.com/package/app-info-parser> (cit. on p. 25).
- [60] URL: <https://www.digitalocean.com/community/tutorials/how-to-create-a-web-server-in-node-js-with-the-http-module> (cit. on p. 37).
- [61] URL: <https://www.ibm.com/cloud/learn/rest-apis> (cit. on pp. 42, 43).

BIBLIOGRAPHY

- [62] URL: <https://developer.android.com/guide/components/fundamentals> (cit. on p. 46).
- [63] URL: <https://medium.com/@adrianstanecki/distributing-and-installing-non-market-ipa-application-over-the-air-ota-2e65f5ea4a46> (cit. on pp. 48, 49).
- [64] URL: <https://www.npmjs.com/package/geoip-lite> (cit. on p. 53).
- [65] URL: <https://create-react-app.dev/docs/deployment/> (cit. on p. 57).
- [66] URL: <https://www.pluralsight.com/guides/deploy-a-react-app-on-a-server> (cit. on p. 57).
- [67] URL: <https://www.cloudflare.com/it-it/learning/ssl/why-use-https/> (cit. on p. 59).