Politecnico di Torino

ICT for smart societies

Master's Degree Thesis

Full stack development of a Credential Manager mobile application

Supervisor
Guido Albertengo

Candidate
Stefano Agnetta

Company supervisors
Federico Arvat
Gabriele Onida

April 2026

To everyone who has left a mark on me throughout my journey.

Abstract

This thesis presents the full-stack development process of *TreasureChest*, a cross-platform Credential Manager application I developed during my thesis program at Lutech, a company specializing in IT consulting and technology delivery. The name *TreasureChest* was chosen to emphasize the value of personal information. The application allows users to securely store and easily retrieve their credentials (such as login data for banking services or social networks) without the need to memorize them. Designed to be intuitive and user-friendly, it targets a wide range of users across different age groups.

To develop the project, I adopted a cross-platform framework. Although the main focus of the thesis program was front-end development, the project has been a full-stack experience: it included the development of a back-end system and a CI/CD section to automate the delivery to both App Store (Apple) and Play Store (Google).

Before deployment, the code is also checked, measuring its quality through static analysis to find eventual vulnerabilities or possible improvements. Finally, functional testing was performed to check if the requirements were effectively accomplished.

Although the project was intended to be curricular with no commercial purpose, *TreasureChest* reached a level of maturity that makes it suitable for real-world deployment, still, in any case, staying in a non-competitive potential level.

Summary

0 Introduction

0.1 Context and motivation

Logging into a website, accessing a bank account, or more generally *authenticating*, requires a *proof* of identity. This proof traditionally consists of two components: a *username* and a *password*, together referred to as a *credential*. The password is the element that must remain secret.

Remembering credentials is crucial for maintaining secure access to sensitive data. A password should be complex enough to prevent guessing or brute-force attacks, and different credentials should never share the same password in order to avoid a single point of failure. As the number of accounts increases, remembering unique and complex passwords becomes impractical, causing necessary to write them down or adopting more advanced strategy necessary.

A **credential manager** system acts as a secure digital replacement for the traditional paper note or post-it, providing a safe and easily usable way to store and retrieve credentials.

0.2 Objectives of a Credentials Manager

A credential manager has the main objective of simplifying the user's life by securely storing credentials and allowing their easy retrieval. It provides a secure environment where authentication data can be managed safely, reducing the cognitive load associated with remembering multiple complex passwords and lowering the risk of insecure user practices such as password reuse, weak passwords or unprotected memorization.

In addition to just credentials storage, a credential manager aims to improve both *security* and *usability*. From a security perspective, it ensures that stored credentials are encrypted and accessible only to the owner thanks to secure authentication mechanisms. From a usability perspective, it improves the user's workflow by allowing credentials to be accessed when needed, across different devices.

The principal functionalities include:

- **Memorization**: securely storing new credentials within an encrypted system
- **Retrieval**: accessing and using stored credentials when authentication is required
- **Editing**: modifying the properties of existing credentials
- **Deletion**: removing credentials that are no longer needed

These represent the core operations of a credential manager, but modern systems often extend them with advanced features such as *password generation*, *password strength evaluation*, *multi-factor authentication*, and *cross-device synchronization*.

0.3 State of the art

As of today, several major Credential Manager systems exist, each with different advantages in terms of UI/UX design, development framework and account features, but they all come with the same primary goal: securely storing credentials and providing users easy access to them. Some of the most notable solutions include:

- **Bitwarden**: An open-source solution with end-to-end encryption, supporting unlimited devices and platforms. Notable features include a secure password and username generator, autofill for web and mobile apps, two-factor authentication (integrated TOTP), password/passkey management, and sharing functionalities. Bitwarden also allows and biometric unlock, and provides business tools like SSO integration and user management.
- **1Password**: Focused on ease of use and security, utilizing dual-key encryption. Offers Watchtower, a security scoring and breach alert system, Travel Mode (temporarily hides sensitive vaults for safe travel), and password sharing. It supports an autofill system, cross-device synchronization, and secure storage for financial information and documents. 1Password provides different plans for individuals, families, and organizations.

- **Google Password Manager**: Natively integrated with Chrome, Android, and the Google ecosystem, enabling password storage, auto-generation, and autofill across all signed-in devices. Credentials are stored encrypted in the user's Google Account. Supports passkeys and cross-device sync, and can also store passkeys. Google's solution prioritizes simplicity over advanced or enterprise features.

TreasureChest, being part of a thesis program, was developed with a formative purpose, following the most instructive and practical development path. It began as a general individual project, and evolved taking inspiration from Bitwarden's main functionalities and design principles.

0.4 Thesis structure

The thesis describes all the step I went through during the development of the project.

The section that compose the thesis are:

1. **Mobile Application Development Evolution**: an overview of the evolution of mobile application development over the years, describing the main approaches and highlighting their advantages and disadvantages.
2. **Requirements Analysis**: an architectural overview of the project, illustrating how the system is designed before its actual implementation.
3. **Technology Stack**: a description of the technologies and technical solutions adopted to fulfill the system requirements.
4. **In Depth System Architecture**: an analysis of the design choices that characterize the system behavior, focusing on architectural decisions and technology integration.
5. **Testing and Evaluation**: a cross-evaluation of the system behavior against the expected requirements, followed by related considerations.

6. **Limitations and Possible Improvements**: a discussion of the system limitations from the perspective of a real-world production scenario, with potential future improvements.

7. **Conclusions**: final considerations regarding both the thesis program experience and the developed project.

1 Mobile Application Development Evolution

Mobile application development has its origins around 1993 with the release of the first smartphone, IBM's *Simon*, but it became popular only after the launch of Apple's first *iPhone* and the subsequent arrival of Google's *Android*-based smartphones. Since then, the world of *apps* has gravitated mainly around these two operating systems.

As smartphones' popularity increased, developers needed to reach more users, making it crucial to build applications for both platforms. At the beginning, this was achieved through straight native development for iOS and Android. Over time, the need to reduce development time and costs led to a *cross-platform application development* approach, which allows developers to create applications for multiple platforms using a single codebase.

Both native and cross-platform approaches have continued to evolve in parallel, each one offering their own advantages on different use cases.

1.1 Native Mobile Development

Although developing applications for a single platform is no longer the only way to deliver mobile software, native development remains a common and often preferred approach for specific use cases.

A native application is built using the official tools and languages provided by the platform's vendor (such as Swift or Objective-C with Xcode for iOS, and Kotlin or Java with Android Studio for Android). This allows direct access to all system APIs and hardware components, ensuring maximum performance and direct integration with the operating system.

Developers typically choose a native development approach when their requirements involve:

- **Performance**: applications that require optimized speed, smooth animations, or real-time processing
- **High OS interaction**: deep integration with platform-specific

features and APIs

- **High hardware interaction**: direct use of sensors, cameras, or low-level device functionalities

The main drawback of native development is its lack of portability: an app developed for one platform cannot directly run on another, and it requires separate development and maintenance efforts. It follows that when performance or software/hardware interactions are not critical, a cross-platform approach is generally preferred to reduce time and cost.

1.2 Hybrid Mobile Development

Hybrid mobile development allows building applications for multiple platforms using a single codebase, representing the main advantage of this approach. It reduces development time and costs, simplifying the product delivery process. In some cases, it may be easier to implement a feature that, to be made it compatible on both the operating systems, has been made simpler.

However, this approach also presents some drawbacks:

- **Reduced performance**: the adaptation between the codebase and the native components can lead to lower execution speed and less efficient resource management
- **Limited API access**: only features supported by both operating systems are available by default, requiring additional effort to integrate platform-specific APIs
- **Hardware compatibility issues**: supporting a broader range of devices increases the likelihood of inconsistencies or untested hardware configurations
- **Higher bug sensibility**: the same codebase may behave differently across platforms, leading to potential instability or inconsistent user experiences

In most cases, these limitations can be mitigated through the use of platform-specific plugins or custom native modules. However, when

performance or software/hardware integration is a key requirement, a native development approach may be more appropriate.

1.3 Comparison: Native vs Hybrid

The choice between native and hybrid mobile development depends on multiple factors such as performance requirements, development time and costs constraints, and target users. The following comparison highlights the main differences between the two approaches.

Aspect	Native	Hybrid
Performance	Optimal	Slightly reduced
Time/costs	Higher	Lower
Access to APIs	Complete	Limited
User experience	Optimal and consistent	May be less responsive
Maintainance	Higher (two codebases)	Lower (single codebase)

Table 1: Comparison between native and hybrid mobile development.

In summary, native development ensures the best performances and can be considered the "high tier" choice. Hybrid development, on the other hand, offers fast delivery and easier maintainance, making it ideal for applications where extreme performance is not a strict requirement and when the target audience is unlikely to perceive significant differences in responsiveness or fluidity.

2 Requirements Analysis

A crucial part of the project was the formalization of requirements, aimed at clearly defining the needs of the end user, the expected system behavior to satisfy them, and the overall design necessary to implement the desired functionalities.

The requirements are formally divided into **user requirements**, **functional requirements** and **technical requirements**.

2.1 User Requirements

This section contains the requirements relative to *what the user needs* from the system, summarized as the functionalities that the application should provide:

- **Sign-up system**: the user should be able to autonomously create a new personal account
- **Login system**: the user should be able to log into their account
- **Vault**: the user, once logged in, should have a *vault* where credentials can be securely stored
- **Master password protection**: the user should be able to access the vault through a user-defined *master password*
- **Biometric unlock**: the user should be able to access the vault using biometric unlock
- **Vault lock**: the user should be able to set a *timeout* after which the vault is automatically locked, and also to lock it manually
- **Credentials management**: the user should be able to *create*, *retrieve*, *update* and *delete* credentials
- **Data encryption**: the vault content (i.e., the user's credentials) should be encrypted end-to-end
- **Clipboard integration**: the user should be able to copy parts of a credential (e.g., username or password) to the clipboard for temporary use, receiving a *Toast* message as feedback

- **Password generator**: the user should be able to *generate* new passwords based on specified constraints (e.g., length, minimum number of special characters)
- **Account password change**: the user should be able to change their account's password
- **Account password reset**: the user should be able to reset their account's password in case it is forgotten
- **Master password change**: the user should be able to change their master password
- **Account deletion**: the user should be able to delete their account permanently
- **Offline support**: once logged in, the user should be able to use the application even without an internet connection, with data synchronization after reconnection

2.2 Functional Requirements

This section describes *what the system shall do* to fulfill the user requirements.

- **Sign-up system**: the system shall allow account creation by providing an *email* and a *password*, which will be used as login credentials; the account shall be activated through a confirmation link sent to the provided email address
- **Login system**: the system shall allow registered and activated users to log in using their *email* and *password*
- **Personal vault**: the system shall provide each user a personal *vault*, associated with their account, to securely store credentials
- **Master password protection**: the system shall protect access to the vault through a *master password*, which is required to decrypt and access stored credentials
- **Biometric unlock**: if supported by the device, the system shall allow the user to unlock the vault using *biometric authentication* as an alternative to manually entering the master password

- **Vault locking**: the system shall automatically lock the vault after a configurable inactivity timeout or when the user explicitly requests it
- **Credentials management**: the system shall allow the user to *create*, *read*, *update*, and *delete* credentials; each credential shall include a *name*, *username*, *password*, and *associated website or description*
- **Data encryption**: whenever an operation involving the vault's content is performed, the system shall decrypt the vault, apply the requested operation, and then re-encrypt it before storage or synchronization
- **Clipboard integration**: the system shall allow the user to copy the selected credential field to the system clipboard for temporary use
- **Toast messages**: the system shall show *toast messages* as feedback for the user when operations (such as copy to clipboard, credential deletion, etc.) are performed
- **Password generator**: the system shall provide a dedicated section for *password generation*, in which the user can specify constraints and a new password can be *generated*, *copied* to clipboard or directly used to create a new credential
- **Account password change**: the system shall allow logged-in users to change their account password after verifying the previous one; the user will have to use the confirmation link that will be sent to the provided email in order to confirm the password change
- **Account password reset**: the system shall allow users that forgot their password to request a reset procedure; the user will have to use the confirmation link that will be sent to the provided email in order to request the password reset
- **Master password change**: the system shall allow logged-in users to change their master password after verifying the previous one
- **Account deletion**: the system shall allow users to permanently delete their account when requested; the user will have to use the

confirmation link that will be sent to the provided email in order to confirm the account deletion

- **Offline support**: the system shall perform all essential credential-related operations for a previously authenticated user; after reconnection, it shall synchronize local data with the remote server
- **Multi-language support**: the system shall adapt its interface language according to the device system language settings, supporting at least *Italian* and *English*

2.3 Technical Requirements

This section defines the technical constraints, so *how the system shall be built* to support the functional requirements.

In general, the system requires a **front-end**, a **back-end**, and a **CI/CD** pipeline mechanism.

2.3.1 Front-end

The front-end consists in a *mobile application* that is the core product that will be used by the final user. To correctly implement all the functionalities, the front-end shall implement:

- **Internet access**, to allow remote communication with back-end
- **HTTP requests**, to ask back-end for user operations
- **AES-256** (in Galois Counter mode) encryption, to secure the vault content
- **Biometric authentication APIs**, to allow the user unlocking their vault through biometric authentication
- **Clipboard APIs**, to allow the user to copy parts of their credentials
- **Toast dialogs**, to notify the user about the result of a requested operation

- **Read/write permissions**, to locally load/save the vault's content (user's credentials) in a text file, providing offline operability
- **Persistent data storage**, to keep session data persistent after closing the application
- **Secure persistent data storage**, to securely store/retrieve the user's master password
- **Multi-language** support, to promote usability for users of different languages
- **Password generation logic**, to allow password generation under some contstraints given by the user

Additional note: being a formative project, to promote the discovery of different functionalities, I opted to store the user's credentials in a local file instead of keep them saved in a secure persistent data storage (which would have been better in terms of security).

Native vs cross-platform project consideration: since all the requirements on front-end fit a cross-platform scenario, an additional fundamental requirement becomes the adoption of a **cross-platform framework**, to generate the code for both Android and iOS operating systems.

2.3.2 Back-end

The back-end's role is to support the mobile application's functionalities, providing all the mechanisms needed to make the user data persistent through different devices. To work as needed, the back-end shall implement:

- **HTTP APIs**, to expose all the useful services to the mobile application
- **SMTP APIs**, to send emails, required for users registration, change of user's password and account deletion
- **Authentication APIs**, to allow user authentication and filter allow/deny requests

- A **database**, to persist user data
- A **database encryption/hashing** mechanism, to securely store user's login credentials
- An **email testing tool**, to provide an easy way of testing that emails work correctly and allow user operations
- An **API documentation tool**, to formalize which APIs are used and ease integration of front-end communication
- A **containerization tool**, to promote tasks separation and to isolate environments

Additional note: being a formative project, all the emails are sent to an **email testing tool**, which served as a *fake smtp server*, instead of being sent to the real email address.

2.3.3 CI/CD

To check the code quality, avoiding possible vulnerabilities, and to ease the release of the final product to both the stores of Apple and Google, the sistem shall implemented a **CI/CD** pipeline mechanism, that to work correctly it needs:

- A **continuous code quality checker**, to detect vulnerabilities in the code and to assert a good implementation of functionalities before the application is built and distributed
- An **automation tool** that automatically build the application from the code, delivering it to both Android and iOS users

3 Technology Stack

Based on the technical requirements, I chose the consequent technologies in order to accomplish all the necessary functionalities.

3.1 Front-end Technologies

3.1.1 React Native

The core front-end technology I chose is **React Native**, which is a *cross-platform framework* developed by Meta. It allows building mobile applications for both *Android* and *iOS* platforms using a single TypeScript (JavaScript) codebase, combining near-native performance with faster development.

React Native adopts a component-based architecture, enabling the creation of reusable UI components and natively integrating device's capabilities through *bridges*, allowing access to hardware components such as biometric sensors, clipboard and local storage.

It's potential is further enlighted by the *atomic design pattern*, which is a developing pattern that organize UI components in five main levels:

- **Atoms**: single-logic and basic UI components, they are the smallest elements (e.g., buttons, text fields, text buttons, etc.)
- **Molecules**: they are made by atoms (or other single-logic elements) combined together, forming more advanced components (e.g., textfield with show/hide text button, text label with on/off toggle, etc.)
- **Organisms**: more complex components that have multiple molecules, allowing different functionalities (e.g., forms containing other components, password strength evaluator form, multiple choice popup menu, etc.)
- **Templates**: consist in the most complete UI component, and represent a screen that can be implemented throughout the application; however, its functionalities are not yet directly implemented

- **Pages**: a template becomes a *page* when the functionalities are fully implemented and integrated in the application's logic

As a cross-platform tool, what React Native offers can be summarized in:

- **Native performance**: UI components are translated into native UI components for the platform the code is built for, resulting in optimal performance
- **Large ecosystem**: a wide range of libraries and community packages make React Native a reliable and tested choice to build consistent mobile applications
- **Hot reload**: enables real-time updates during development without rebuilding the entire application, improving productivity and development speed

The principal mobile technology adopted by the Lutech Practice in which I had my thesis program is React Native, and that has been the main reason for choosing it as the cross-platform framework for the project. The benefits of using React Native over Flutter are mainly related to its use of *native UI components*, making it suitable for applications that want to "fit" the operating system's visual design; consequently, being native components, they are rendered without any performance cap, typical of canvas-based rendering. In addition to that, React Native is a more stable and widely adopted framework compared to Flutter, which makes it a more logical choice for applications, such as TreasureChest, that need a certain level of robustness and reliability.

3.1.2 Expo

Expo is a framework that acts as a superstructure for React Native projects: it pre-implements a base structure that helps the management of dependencies and directly manages native code generation and applications building, interacting with Android SDK and iOS SDK.

Key features of Expo include:

- **Simplified setup**: it configures native sections and complex dependency linking

- **Expo SDK**: offers a comprehensive set of APIs for different native functionalities (e.g., local authentication, file system, clipboard, etc.)

Although Expo is a standard in React Native applications, it is worth to underline that its adoption allowed me to focus more on project development instead of building its structure to make it work. It has been a way to avoid "reinventing the wheel".

3.1.3 Redux (with RTK)

Redux is a JavaScript library that is mainly employed to manage the *global state* of an application, so what is globally accessible throughout the whole application during its execution.

This is particularly useful to handle pieces of information that are used in multiple section of the application, and retrieving them every time they are needed may result in excessive operations. Global state is not meant to be over-used: the need of having some information globally shared should be directly proportional to the complexive effort needed to retrieve it (e.g., reading the vault's content may be computationally expensive, so doing it every time it is required could unnecessarily slow down the application's use experience) and inversely proportional to the number of times it may change through time (e.g., username is unlikely to change frequently, so having it in the global state reduces the number of time the same information will be retrieved).

While Redux is the core library that implements the global state, **Redux Toolkit** is the package that is intended to be the standard way of implementing Redux' in an application. It simplifies the logic and reduces boilerplate code, by exposing the core functions and patterns to implement global state and its related benefits.

Redux operates through these key components:

- **Slices**: represent independent pieces (slice) of information, from the global state, that can be considered *independent* from others (e.g., the slice related to user session is independend from the password generation one)

- **Reducers**: functions that define how the state changes in response to some operations relative to the global state (e.g., setting a property such as the colors scheme)
- **Thunks**: a thunk is mainly a specialized reducer designed to perform *asynchronous logics* (e.g., reading from a local file and setting the user's credentials)
- **Selectors**: they are used to access information from a slice of the global state (e.g., username selector)

To make global state *persistent* when closing the application, a library called **Redux Persist** is employed.

In the project, adopting Redux has been fundamental to correctly implement global state management for several sections: user information and authentication (tokens), vault credentials, password generation settings, and inactivity detection.

3.1.4 RTK Query

RTK Query is an optional addon included in the *Redux Toolkit* package. It is a powerful data fetching and caching tool designed to simplify common cases for loading data from remote into an application. Its functionalities of data fetching, caching, and automatic revalidation are built on top of the other APIs in Redux Toolkit.

It shines when adopted to implement the *network layer* for an application, to organize the communication with the back-end endpoints, offering a well-structured approach. It automatically handles caching, refetching and state invalidation, drastically reducing manual state management.

RTK Query has some key concepts:

- **Endpoints**: define the API operations, such as *queries* (data retrieval) or *mutations* (data modification). Each endpoint corresponds to a specific back-end route.
- **Queries**: represent *read* operations (e.g., fetching the user's credentials or profile). The result of a query is automatically cached and reused.

- **Mutations**: represent *write* operations (e.g., login, registration, credential creation). They can automatically trigger cache invalidation or data refetching.
- **Cache management**: RTK Query automatically caches the responses of queries and provides control over invalidation and refetching.
- **Auto-fetching and revalidation**: data can be automatically refetched when a component mounts, when the application regains focus, or after a mutation changes related data.

In the project, RTK Query has been a key element for the organization of all communication between the front-end and back-end, defining endpoints for operations such as user authentication, account management, and vault synchronization.

3.1.5 SecureStore

SecureStore is a library from Expo that provides a way to encrypt and securely store key-value pairs in the project-dedicated local filesystem. It is also designed to provide a persistent data storage system, so that data is not lost when the application is closed.

It acts as a bridge between React Native and the native secure storage systems used in the two platforms: *SharedPreferences* combined with *Android Keystore* for Android, and *Keychain Services* for iOS.

SecureStore is particularly suited for securely persisting small sensitive pieces of information—such as authentication tokens or encryption keys, that must remain accessible across sessions without repeatedly requiring user input.

I used SecureStore library for memorizing the master password, that is frequently used to decrypt the vault across a single session through biometric authentication, but in a real project it could have been employed for credentials memorization too.

3.1.6 LocalAuthentication

LocalAuthentication is an Expo module that allows the use of the Biometric Prompt (Android) or FaceID and TouchID (iOS) to authenticate the user with a fingerprint or face scan, if available by the used hardware.

In this project, I implemented it to enable *fingerprint authentication* as an alternative method to unlock the vault, to offer a faster and more convenient access mechanism while maintaining security. Before performing a biometric check, the module verifies whether biometric hardware is available or not, and if the user has configured at least one fingerprint. If these conditions are not met, the system will fall into master password authentication.

3.1.7 Toast Messages

A **toast message** is a small popup used as light notification for the user, usually shown when an operation is completed (successfully or not), without the need of user interaction.

Toast messages are basically a text within a View shown on top of the screen, and in this project are implemented through a React Native module called *React Native Toast Message*. They are particularly useful to give feedback after credentials creation/update/deletion or when some content has been copied to the clipboard.

3.1.8 Expo Clipboard

Expo Clipboard provides an interface for getting and setting Clipboard content on Android and iOS.

Although both *copy to clipboard* and *paste from clipboard* flows are enabled, in the project the content is only **copied to clipboard**.

When some content is copied, the application gives feedback to the user through *toast messages*.

3.1.9 Cryptography

The cryptography layer of the project is built on top of two main modules: **react-native-aes-gcm-crypto** and **ExpoCrypto**.

The first one is used to implement **AES-256** (in Galois Counter mode), and consequently the functions to *crypt* and *decrypt* a string, which are necessary to securely store the vault's content (i.e., the user credentials).

The second one allows the implementation of a *hashing function* (based on SHA256) and a *random numbers generator*, functions that are used across the application for various purposes (e.g., storing the master password, password generator).

3.1.10 i18next

i18next is a JavaScript internationalization framework that provides management of translations and localization within an application.

The framework follows a *key-value* mapping approach, where all the available string are referenced by their keys. Each supported language has its own *JSON* file containing the same set of keys, but with language-specific values. At runtime, the framework selects the correct JSON file based on the user's language preference (or, eventually, on the developer's choice), returning the correct translated text.

To have a typed-mechanism for retrieving the application's strings I developed, on top of this library, a super-structure that associates a *typescript interface* to the strings structure, so that when a string must be retrieved it is possible in advance to know how it is called inside the JSON file, instead of having to manually find it.

3.2 Back-end Technologies

3.2.1 Docker

Docker is a platform designed to simplify the development, deployment, and execution of applications through the use of *containers*. A container is a lightweight, isolated environment that contains an

application and all its dependencies all together, ensuring that it can run correctly across every environment and operating system that can run Docker.

The core components of a Docker ambient are:

- **Images**: immutable template that contain everything needed to run an application (i.e., system tools, dependencies, code, etc.)
- **Containers**: running instance of images; they represent the execution environment of an image, and they are isolated from the host system while being able to interact with it through specifications. Containers can be easily created, stopped or removed without affecting the rest of the system
- **Volumes**: persistent storage units used to persist data of a container, independently from the container's lifecycle. They are basically what is persistent of a container (database, useful files, media, etc.)

Docker relies on two main components:

- **Dockerfile**: a configuration file that defines the environment, dependencies, and build instructions required to create a container image.
- **Docker Compose**: a tool for defining and managing multi-container applications using a single *YAML* configuration file, allowing easy orchestration of multiple *services* (e.g., back-end with database and fake SMTP server together).

The main advantages of adopting Docker are:

- **Environment consistency**: the same container image can be executed across different systems without configuration conflicts.
- **Simplified deployment**: containers can be quickly started, stopped, or replaced.
- **Isolation**: each container runs in its own environment, preventing dependency conflicts between different services or other containers.

In this project, I used Docker to containerize these components:

- Back-end
- PostgreSQL (database)
- Mailpit (fake SMTP server)
- SonarQube (code quality checker)

Adopting Docker allowed me to easily orchestrate how the single components must interact with each other.

3.2.2 PostgreSQL

PostgreSQL is an open source *relational database*, based on SQL standards. It is designed to efficiently handle structured data while supporting complex queries and advanced data types.

Its core components include:

- **Tables**: the principal data structures that organize information using a row-column approach. Each table corresponds to a specific entity (e.g., users, vaults, etc.)
- **Schemas**: logical namespaces that group database objects (tables, views, functions)
- **Indexes**: data structures that improve query performance by optimizing access to frequently searched fields
- **Constraints**: rules that promote data integrity (e.g., primary and foreign keys, unique or not null constraints)

I used PostgreSQL as the main database to persist user-related data, such as account information, credentials, and session states, ensuring data consistency and integrity. The setup follows PostgreSQL running in a Docker container. This configuration allows for easy deployment, isolation, data persistence through *Docker volumes*. The back-end server, itself running in a dedicated container, communicates with the PostgreSQL container through a defined network, isolating the database from the external environment while maintaining accessibility.

I adopted a SQL database instead of a non-SQL one because of the nature of the data the application handles: being always with a predefined structure, the adoption of a structured database has been a natural choice.

3.2.3 Spring Boot

Spring Boot is an open-source Java framework built on top of the *Spring Framework*, designed to simplify the development of server applications. It provides a solid setup that reduces the need of an excessive configuration, enabling developers to create and run applications quickly reducing boilerplate code.

Spring Boot's main features include:

- **Auto-configuration**: automatically configures the application based on the dependencies present in the project, reducing the need for manual setup
- **Dependency management**: integrates with *Spring Initializr* and *Maven/Gradle* to simplify dependencies and project setup

In this project, Spring Boot is used as the core back-end framework, providing:

- A set of **RESTful APIs** to handle communication with the mobile application
- Integration with a **PostgreSQL** database for data persistence
- Support for user management and authentication with **JWT**
- Connection with the **Mailpit SMTP server**

The interaction with the database resources is implemented through **JPA (Jakarta Persistence API)**, to simplify the construction of small operations and reducing explicit boilerplate SQL queries.

Moreover, it is worth to specify that the language I adopted for the core back-end is **Kotlin**. Although it is common to use Java, being the standard adopted in Spring Boot based applications, I chose Kotlin for

formative reasons, to maintain the line with a programming language usually adopted in the development of Android applications.

3.2.4 JWT (JSON Web Tokens)

JWT (JSON Web Tokens) is an open standard that is widely adopted for authentication and authorization in modern web and mobile applications, allowing stateless user sessions without the need to store session data on the server.

A JWT consists of three parts:

- **Header**: specifies the type of token and the signing algorithm used (e.g., HS256)
- **Payload**: contains the *claims*, i.e., statements about an entity (typically the user) and additional metadata (e.g., user ID, expiration time)
- **Signature**: ensures the token's integrity and authenticity, generated by signing the header and payload with a secret key (stored on the back-end itself)

The resulting JWT is a string in the following format:
`header.payload.signature`.

In this project, JWTs are used to authenticate users after a successful login. The back-end issues a signed token that the client stores locally, that will be attached to subsequent requests as a proof of its identity. Each token includes an expiration time to enhance security and reduce the risk of unauthorized access.

3.2.5 Mailpit

Mailpit is a lightweight *email testing tool* designed to capture outgoing emails during the development phase. It acts as a *fake SMTP server*, meaning that it intercepts emails sent by an application without actually delivering them to real addresses. This allows developers to verify email content and formatting without the risk of sending test messages to end users.

Mailpit consists of two main components:

- **SMTP server**: receives emails from the application as if it were a real mail server
- **Web interface**: provides a graphical dashboard where all the captured emails can be viewed and inspected

In this project, Mailpit is used to handle all email-based features:

- User registration confirmation
- Password reset requests
- Account deletion confirmation

Mailpit runs as a dedicated Docker container. The back-end service connects to it through its exposed SMTP port, while, for simulation purpose, the web dashboard is also exposed so that the user can perform the required operations as if they were on a real email account.

Since the main focus of this project's back-end adoption was to support the mobile application, implementing a real SMTP server would have been out of scope. Moreover, for testing purposes, a fake-SMTP server allows all the main functionalities needed to implement the mail functionalities inside the system. This is the main reason why I decided to adopt a fake SMTP server.

3.3 DevOps Technologies

3.3.1 Jenkins

Jenkins is an open-source *automation tool* designed to support *DevOps* practices, such as **Continuous Integration** and **Continuous Deployment** (*CI/CD*) processes. It allows automatic building, testing, and deployment of applications, helping reduce human errors and improving efficiency in the software delivery pipeline.

The main components and concepts include:

- **Pipelines**: define automated workflows as code, specifying build, test, and deployment stages
- **Jobs**: represent individual automation tasks (e.g., build a project, run tests, deploy an artifact)
- **Credentials**: authentication elements required during the process that must be kept secret. They can take different forms, such as *username-password* pairs or *tokens*, and can be either *global* or *local*
- **Secrets**: variables whose values are securely stored and accessible only within their pipeline or job context
- **Plugins**: extend Jenkins' functionality, enabling integration with version control systems (e.g., GitHub), build tools, testing frameworks, and deployment services

Jenkins is commonly adopted to automate back-end development process, but can also be employed for the automatic deployment of mobile applications, resulting particularly useful in cross-platform projects, as a single code-base will produce the applications for both Android and iOS.

A specific hardware requirement for producing applications for the Apple's mobile ecosystem is running the building process on an Apple machine with **MacOS**: the required SDK is not available on other operating systems.

In this project, Jenkins's use is summarized in:

- Automatically building the mobile application project after code changes are detected
- Running a static code analysis tool to ensure code quality
- Packaging and distributing application to both Android and iOS stores

3.3.2 SonarQube

SonarQube is an open-source platform for **code quality inspection** and **static analysis**. It analyzes source code to detect issues related to

bugs, *code smells*, *security vulnerabilities*, and *maintainability problems*, providing developers with feedback before deployment.

SonarQube integrates seamlessly into CI/CD pipelines and supports a wide range of programming languages, including *JavaScript*, *TypeScript*, *Java*, and *Python*. It can be configured to automatically run at each build, ensuring that code meets predefined quality standards, which can be configured depending on the user needs (e.g., a security application will need to have a close-to-zero score of security vulnerabilities).

The main components of SonarQube are:

- **Scanner**: the client-side tool that analyzes source code and sends the results to the SonarQube server
- **Server**: processes and stores the analysis results, providing a web interface to visualize metrics and reports
- **Quality profiles**: define coding rules and thresholds that determine when code passes or fails quality checks (e.g., code duplications, maintainability code practices, etc.)
- **Quality gates**: enforce standards by marking builds as failed if the quality thresholds are not met

In this project, SonarQube runs as a Docker container and is integrated into the Jenkins pipeline to:

- Perform static analysis on the codebase before build and release
- Detect potential security vulnerabilities or performance issues early in the development cycle
- Ensure code maintainability and consistency across the project

4 In Depth System Architecture

4.1 Front-end

The project's front-end is the *mobile application* used by the user; before starting the development of all the screens, I designed a navigation graph (Figure 1) to have in advance a clear understanding of how the screens must be linked together, and which type of data they exchange each other.

All the screens inside the application are build following the **Atomic-design pattern**: a screen is a *page*, which is the implementation of a *template* with all the data fields populated. The navigation links are developed through the project's dedicated section, separated from the UI/UX domain.

Each macro-section of the application will be explained its the relative paragraph below.

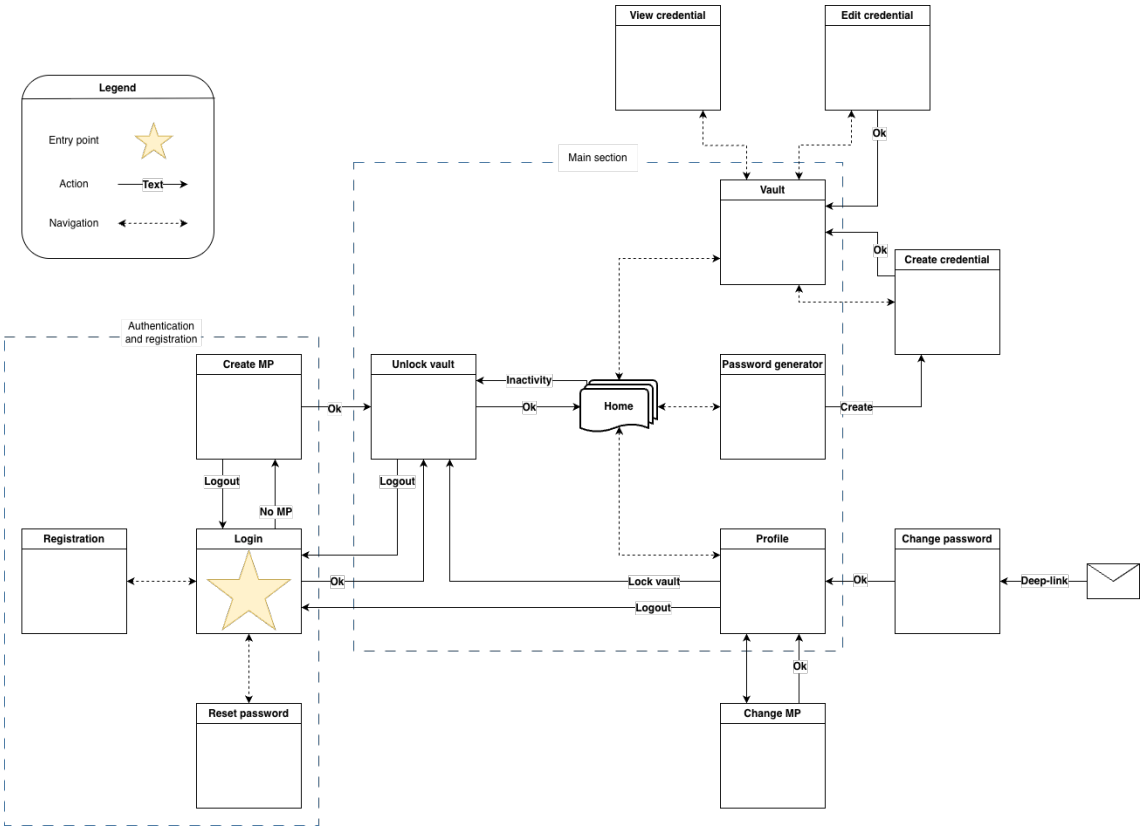
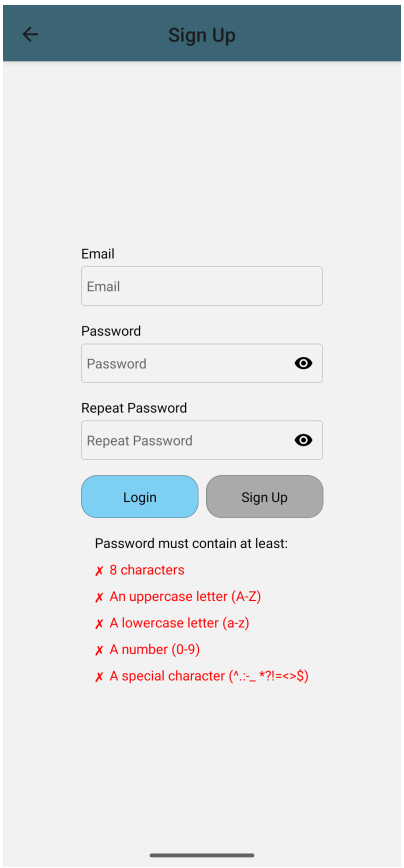


Figure 1: Navigation graph of the mobile application

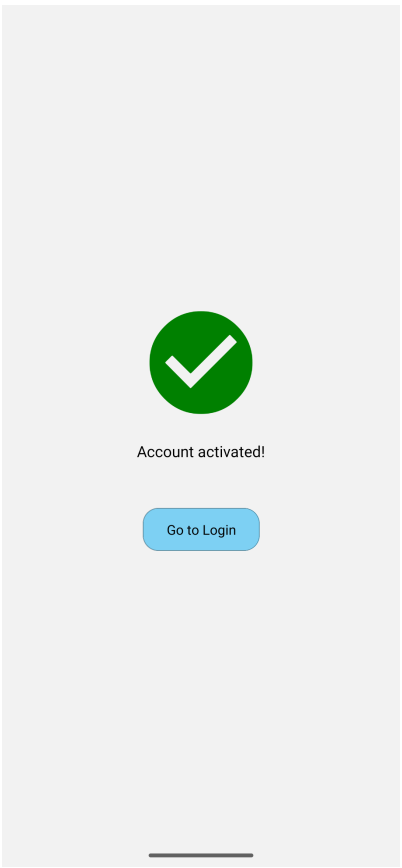
4.1.1 Authentication

In this section, everything concerning the authentication flow will be explained.

Account creation When the user launches the app for the first time, they must create a new account through the **Sign-up** screen (Figure 2a), providing an *email* and a *password*.



(a) Sign-up screen



(b) Activated account screen

I set some *constraints* to the password, to ensure a minimum level of security:

- **Length**: the password's length must be of at least *8 characters*
- **Uppercase letters**: at least *one* uppercase letter is required
- **Lowercase letters**: at least *one* lowercase letter is required
- **Numbers**: at least *one* number is required
- **Special characters**: at least *one* special character is required

These constraints ensure that a brute-force attack would require at least a^N attempts (where $N = 8$ is the minimum password length, and a is the number of possible characters), making a guess theoretically infeasible.

Submitting the form triggers the back-end to send an email to the address provided by the user. The email contains a **deep-link** that points to a specific application screen and includes the *user unique id* as a parameter. When the user opens the deep-link, the application automatically sends an **account activation request** to the back-end.

If the operation is successful, the user's account is activated (screen in Figure 2b), enabling them to log in to the application using the credentials defined during registration.

Login An activated user can log into the application providing their *email* and *password* (Figure 3), and if the login process is successful, the next screen is shown: master password is asked to unlock the vault if it was previously created, otherwise the user will be asked to create it through the relative screen (Figure 6).

Meanwhile, under the hood, the **JWT exchange** process is performed (detailed process is explained in the specific section 4.1.2). The user receives *access* and *refresh* tokens, crucial for their session across the whole application usage.

Password reset If a user forgets the password, they can request a reset by clicking on *Forgot password*: this opens a screen where the user is asked to enter their email address to which the deep-link to effectively allow them to reset the password will be sent.

The email contains a **deep-link** that includes the *user unique id* as a parameter and points to a dedicated application screen where the user can define the new password. Once the form is submitted, if the passwords inserted into the two textfields match, the back-end updates the user's credential. The user can then log in using the new password.

Security note: the request to reset a password is *not authenticated*, which means that anyone in possession of the URL sent by email (i.e., the deep-link) is able to perform the password reset operation. To

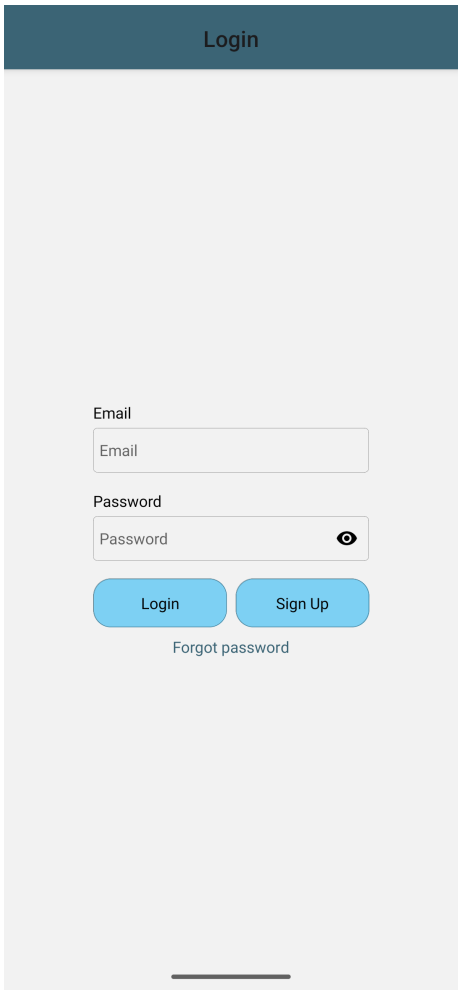


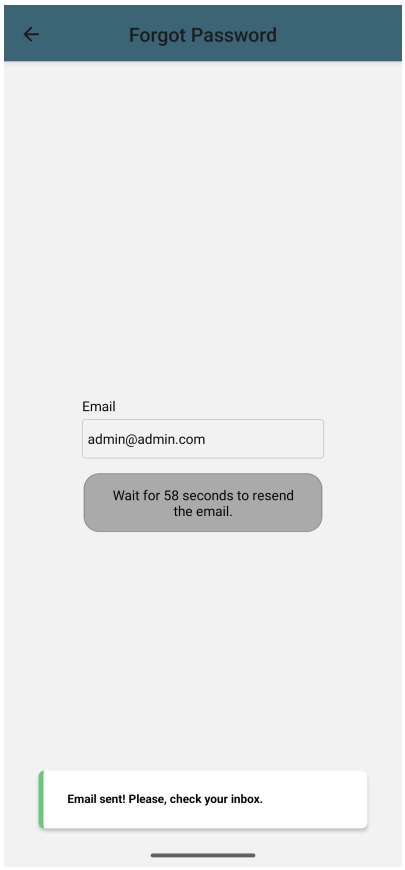
Figure 3: Login screen

mitigate this risk of unauthorized password changes, the back-end associates a *validity token* to each password reset request, that expires after 15 minutes. Once the token is expired, the deep-link becomes invalid and the back-end rejects any attempt to reset the password using that URL.

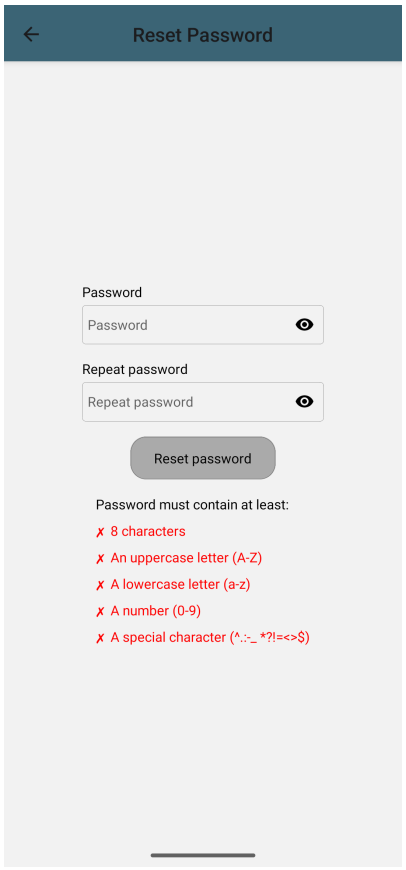
4.1.2 User session

A user that successfully logs into the system receives a pair of JWT tokens, issued by the back-end, called **access token** and **refresh token**. These tokens allow the user to maintain an *authenticated session*. They both have a *duration* defined by the issuer (i.e., the back-end), which is typically significantly longer for the refresh token.

The access token is used to prove the user's identity, and is attached to the requests sent by the application. When it expires, the user must



(a) Requesting a password reset



(b) Resetting the password

perform a **refresh request** using the refresh token to obtain another new access token.

When the refresh token expires, the user must log in again to obtain the new pair of tokens and start a new session.

Both the tokens are discarded when the user logs out.

The complete workflow is shown in Figure 5.

Token handling Once received from the back-end, both the **access token** and **refresh token** are securely stored inside **SecureStore**.

Automatic authenticated request When a request is sent, *Redux' RTK Query* module handles the whole communication flow with the back-end: before sending the request, if stored in memory, the *access token* is injected into the **authorization header** ("Bearer: ¡token¿").

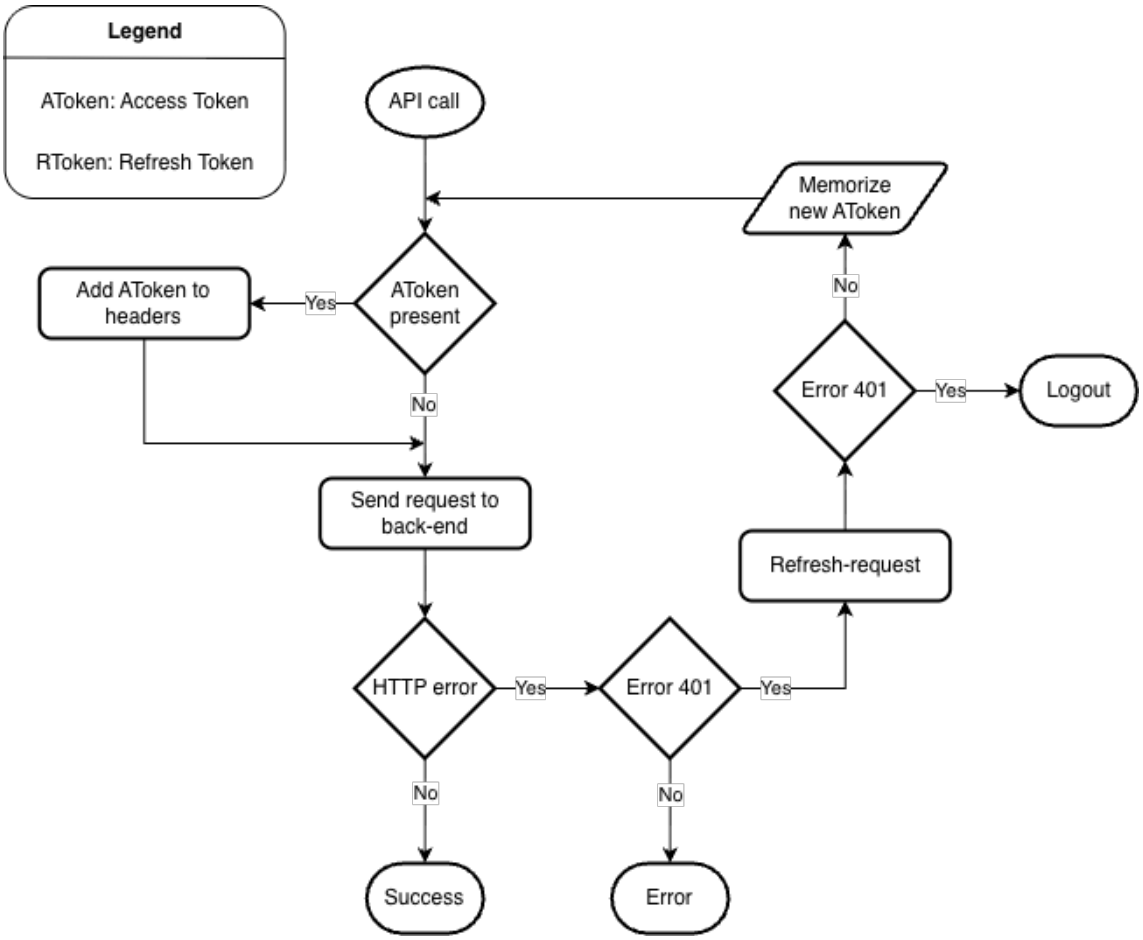


Figure 5: Authentication lifecycle scheme

Automatic refresh request As the response is received by the mobile application, RTK Query checks the result: if it is an **unauthorized (401)** (and the access token was attached to the associated request), it automatically sends a **refresh request** to obtain the new access token, and then the request is sent again with the new token attached; if the refresh request fails, it means that the refresh token is expired, which causes the application fo remove the stored tokens and the logout is then forced.

Session termination Logout triggers a complete cleanup: the two tokens are removed from SecureStore, and the application invalidates the local session.

4.1.3 Master Password Creation

To guarantee the *confidentiality* of the user's stored credentials, all data inside the vault is kept **encrypted**, meaning that nobody is able to read them without knowing the *secret key* used in the encryption process. In this context, the secret key is called **master password**.

After the login, before being able to use the system, the user must create their master password (Figure 6), which has to be kept *secret* to protect the credentials. Once defined, it is securely stored using the **SecureStore** module, under which the application's security is based.

Whenever the vault is modified, the master password is used as a decryption key, changes are applied, the updated data is encrypted again and saved into the filesystem, ready for the synchronization with the remote server.

The master password can be successively changed from the user's *profile* page.

If the user forgets the master password, the vault's content is inaccessible. This is an intentional design choice: the system provides **no mechanism to retrieve the master password**, ensuring that no unwanted user can unlock the vault without knowing it. On the other hand, this forces the user to remember it without the possibility to recover it. For this reason, the constraints for choosing the master password are less strict (compared to that for the login password):

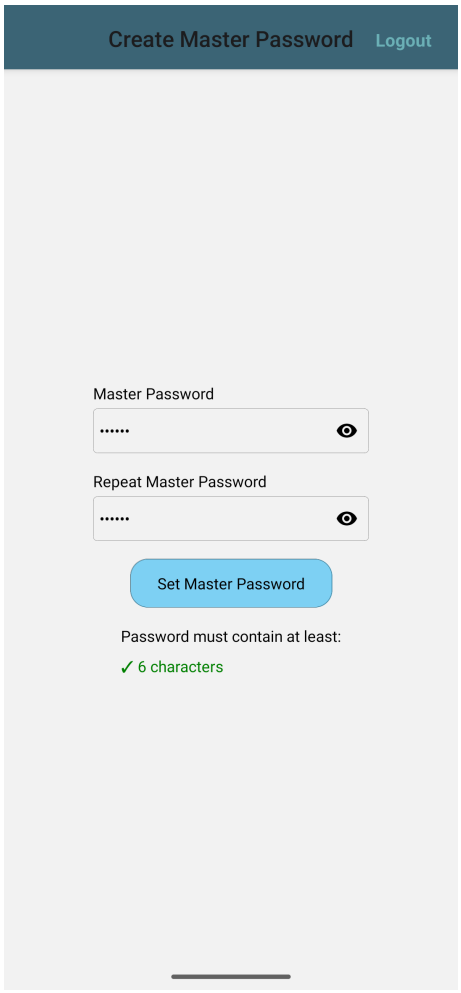


Figure 6: Master password creation screen

only a minimum length of **6 characters** is required, allowing the user to select an easier secret to remember.

4.1.4 Unlocking the Vault

After logging in, the user is required to provide the master password to decrypt the vault and access its content (Figure 7). This behaviour ensures an additional layer of protection that remains active throughout the user's session.

If **biometric authentication** is available and enabled, the user may choose to unlock the vault using their **fingerprint** (Figure 8b). In this case, a system prompt requesting the fingerprint is displayed; the user may also trigger it manually by tapping the circled-fingerprint button at the bottom of the form.

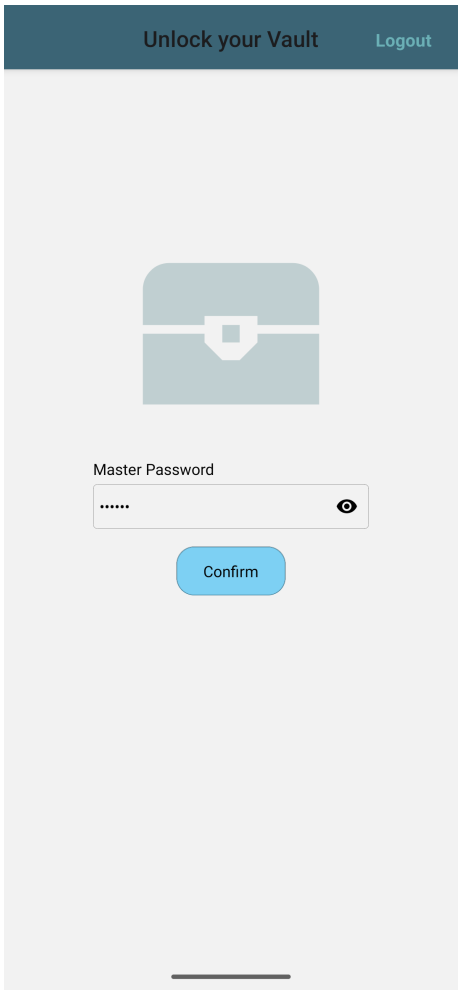


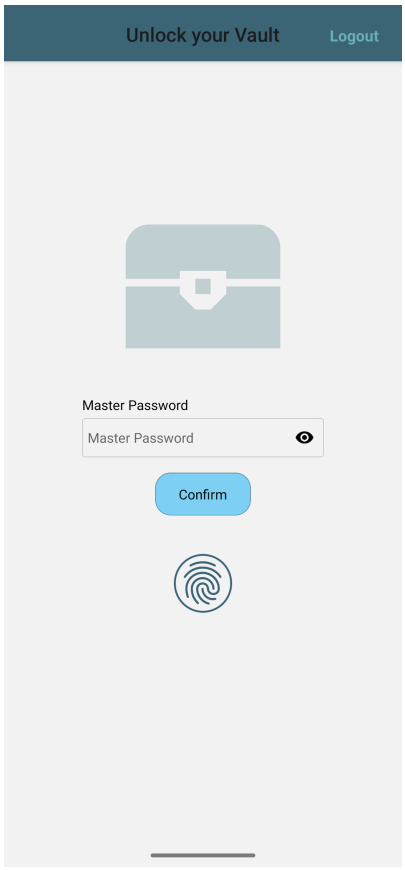
Figure 7: Unlocking the vault

Even though the user is not explicitly providing the master password, it is still used internally, as it has been securely stored using SecureStore.

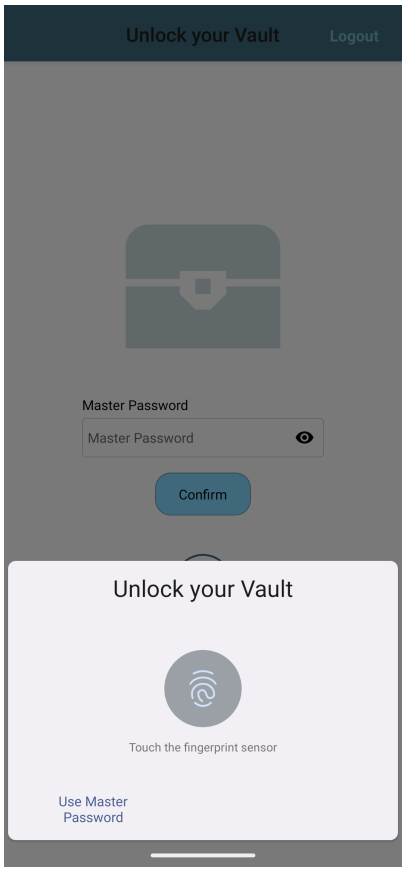
As a further layer of security, I implemented an automatic locking system: the vault is locked when **inactivity** is detected. While the application is in focus, every user interaction resets an internal *timer*. If the timer reaches its timeout, the vault is automatically locked. This ensures that if the user forgets to manually lock the vault or to lock their device, the credentials are less exposed to unauthorized access. The user can set a custom timeout in the *Profile* page.

4.1.5 Credentials Management

Once the vault is unlocked, the user can access the system. The home screen is a **tabbed view**, whose tabs are:



(a) Vault unlocking with biometric authentication enabled



(b) Unlocking the vault using biometric authentication

- **Vault** tab: where the credential management has its core
- **Password Generator** tab: the user can generate new passwords under provided constraints
- **Profile** tab: the user can customize their account settings (e.g., their personal password)

The **Vault** page (Figure 9) is where all credential-related operations are performed.

Credentials management is the core functionality that the application offers, as it represents what a credential management application is: memorization and retrieval of user's credentials.

What the user can user can do, is summarized in:

- **Creation**
- **Visualization**

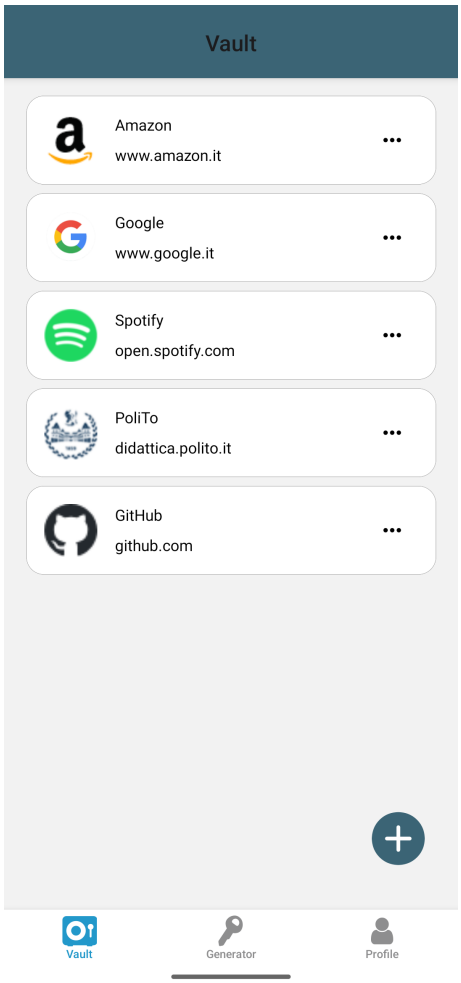


Figure 9: Vault tab with a list of credentials

- **Modification**
- **Deletion**

Create a new credential By clicking on the *floating action button* located at the bottom-left corner of the application, the user can access the credential creation form that asls to fill four fields:

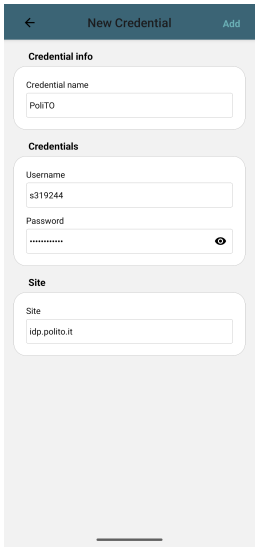
- A **credential's name**, which is the name used to identify the credential
- A **username**, which is essentially the property used to identify the user inside the context the credential is going to be used
- A **password**
- A **site** that may be a website URL or any information useful for the user to associate their credential

Once the fields are filled (Figure 10a), the new credential can be added to the vault.

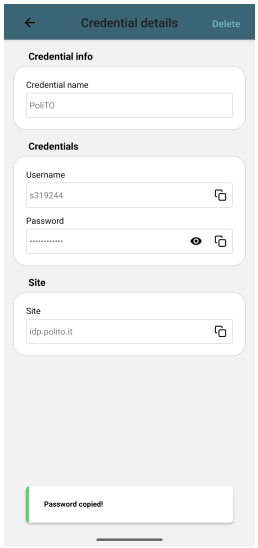
Visualize a credential To visualize the details of a credential, the user selects it from the list (Figure 9). The credential's fields are shown in the view that appears (Figure 10b), where the user can *copy* any value to the *clipboard*.

Modify an existing credential To modify an existing credential, the user opens the *menu* (three-dots icon next to the item in the list) and select the edit option. This opens an editing screen (Figure 10c) where the credential's fields can be updated. Once finished, the changes must be saved.

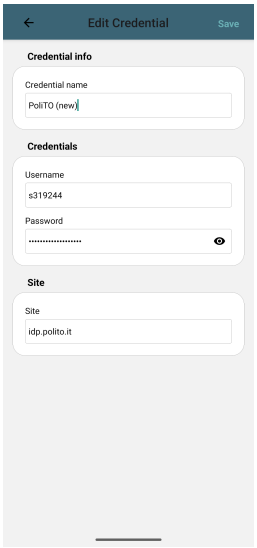
Delete a credential A credential can be deleted from its visualization screen. The operation is irreversible. Once a credential is deleted, the user is returned back to the Vault screen (Figure 10d).



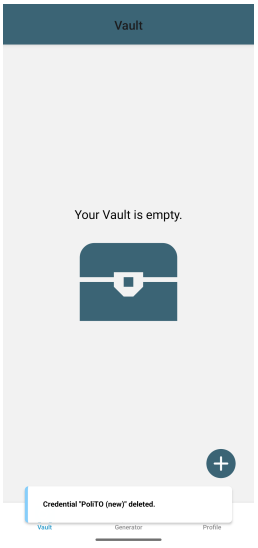
(a) Credential creation



(b) Visualizing a credential



(c) Modifying an existing credential



(d) The user has deleted a credential

4.1.6 Password Generator

In the **Password Generator** tab (Figure 11), the user can generate new passwords with some customizable characteristics. This feature is particularly useful when a service requires a password that satisfies

some specific constraints such as length, minimum number of digits or inclusion of special characters.

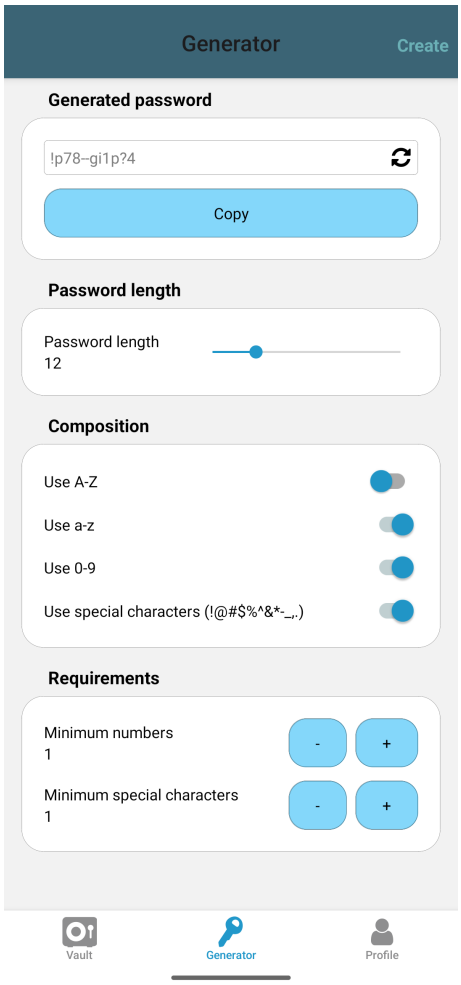


Figure 11: Password Generator tab

Password length The required password length can be set using a *slider* component. The allowed range of lengths goes from **6** to **32** characters.

Composition The user can specify which types of characters are allowed in the generated password through *toggle* controls:

- **Uppercase letters**
- **Lowercase letters**
- **Numbers**
- **Special characters** (`!@#$%^&*-_,.`)

Requirements The user may want to ensure that generated passwords must include a minimum number of specified character chategories:

- Minumum number of **digits**
- Minumum number of **special characters**

These values can be tuned using *"+"* and *"-"* buttons. If the sum of the required characters is bigger than the selected required password length, the latter is automatically adjusted so that it is *at least* equals to the sum of the requirements.

Generation logic I implemented password generation procedure as follows:

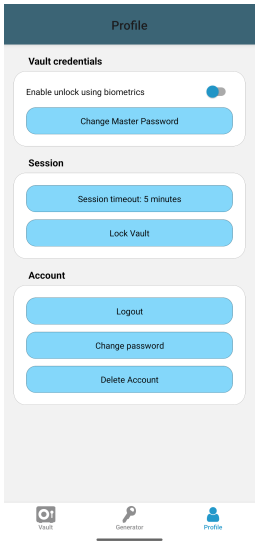
1. An *empty string* is initialized
2. The specified number of minimum *digits* are picked at random from the possible digits (i.e., numbers from 0 to 9) and are appended to the initial string
3. The specified number of minimum *special characters* are picked at random from the possible special characters and are appended to the previous string
4. The remaining length is filled picking at random the characters specified in the *composition section*
5. The final string is shuffled to ensure randomness
6. The string shuffled, i.e., the generated password, is shown in the related field

The user can regenerate new passwords using the *refresh button* next to the field where the generated password appears. They may decide to *copy to the clipboard* the new password, or use it for a new credential clicking on *Create* text button at the right of the page title.

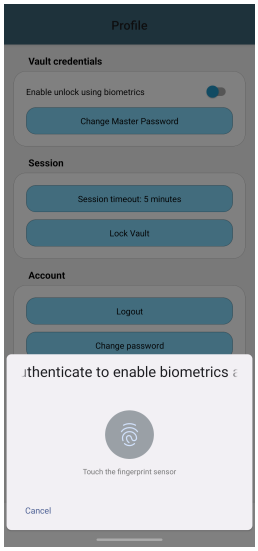
The password Generator Page can be considered one of the key "bridges" between the user and the system, as it provides a fast and secure way to generate strong passwords answering to the user's needs.

4.1.7 App Settings

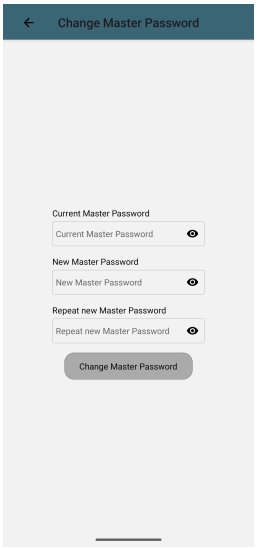
The user can adjust application usage preferences, related to both **vault** and **session**, within the **Profile** tab (Figure 12a).



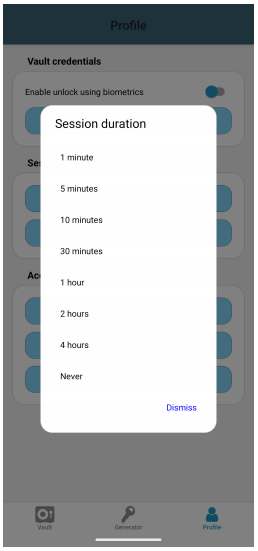
(a) Profile tab



(b) Enabling biometric unlock



(c) Changing master password



(d) Setting session timeout

Vault settings If *biometric authentication* is available on the device, the user can enable vault unlocking through biometrics. The only biometric method supported by the system is the **fingerprint**, which is firstly asked to the user, via a system prompt (Figure 12b), in order to verify the user's identity.

The user can also change the *master password* by clicking on the relative button, which opens a page with a form (Figure 12c) requiring the current master password and the new one (repeated for confirmation). What is important to underline again is that if the master password is forgotten, it cannot be recovered and the credentials stored in the vault are permanently lost.

Session settings With respect to the *session*, the user can configure a **timeout** by selecting one of the predefined values (Figure 12d). Additionally, if needed, the vault can be manually locked, forcing the user to enter the master password or to provide its fingerprint (if biometric authentication is enabled).

4.1.8 Account Management

The user can manage their account through the functionalities available in the *Profile* tab.

Specifically, the user can:

- Change the account password
- Delete their account

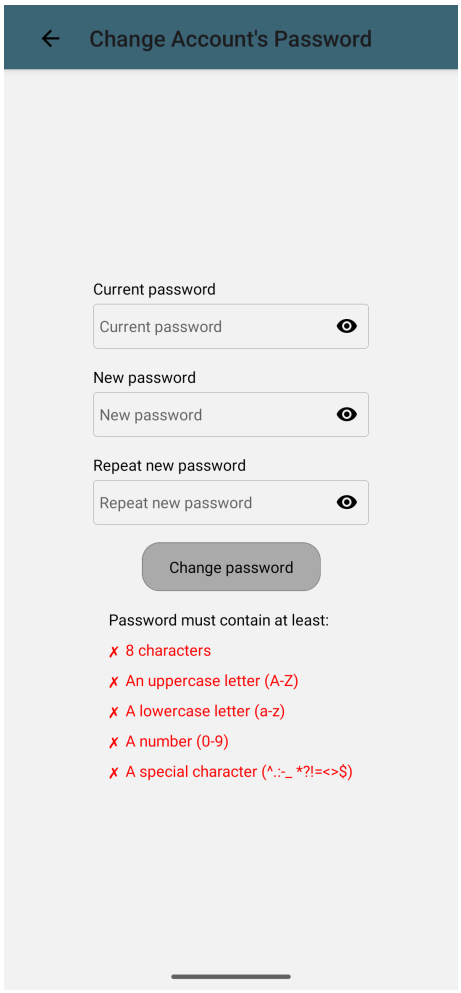


Figure 13: Changing the account password

Changing the password To change the account password, the user must request the operation using the corresponding button. An email containing a *deel-link* is sent to the user's email. Once opened, it redirects the user to a screen (Figure 13) that requires to enter the current password and the new desired password (repeated for confirmation).

Submitting the form triggers a request to the back-end that, if the current password is entered correctly, will updated the user's credential.

Deleting the account As a last resort, the user may decide to permanently delete their account with all the associated credentials. By clicking on the corresponding button, an email containing a confirmation *deep-link* is sent to the user. When opened, a **delete account** request is sent to the back-end

If the operation is successful, the user's account is removed from the database, the user is logged out from the application, and all the locally stored data is deleted (i.e., the credentials and the authentication tokens). This operation is **irreversible**.

4.1.9 Multi-Language Support

To reach a wider range of users, it is fundamental to integrate **multi-language** support into an application.

In TreasureChest, multi-language is integrated through adaptation to system's settings: the application language reflects the system language at runtime. Moreover, a *fallback mechanism* is implemented: if an unsupported language is set, the application will provide content in English by default.

Although the project only supports **italian** and **english**, it would be possible to implement more languages by simply adding new translation files.

4.1.10 Offline Support

The application is designed to guarantee usability under any network condition. Therefore, it is expected to work in the absence of internet connection or even when the back-end is temporarily unreachable (e.g., due to maintenance operations). The user must be agnostic on what is happening under the hood.

Providing resilience against **offline conditions** requires the adoption of a strategy that allows the user to perform most operations as

if the application were offline. In the context of TreasureChest, this primarily means **preserving local access to the vault** and **ensuring synchronization** with the remote server once connectivity is restored.

The system must perform **data synchronization** to keep credentials consistent, even in case of modifications performed while offline, regardless of whether they were performed on the same device or on different ones.

There are several ways to silently handle the absence of internet connection; the one I decided to adopt is the **offline-first** strategy: all vault data is stored encrypted locally, and represents the primary source of truth while the application is offline. During this phase, the user can freely create, view, modify or delete credentials without requiring any interaction with the back-end. The strategy aims to preserve the *most up-to-date data*. To allow this behaviour, both the vault and each individual credential must keep track of a **last modification timestamp**, which is updated every time a change occurs.

The synchronization strategy is performed every time the system fetches the vault from the back-end. The procedure is the following:

1. The remote vault is retrieved from the back-end.
2. The *last modification timestamps* of the local and remote vaults are compared. The the most recently updated one is selected as the reference.
3. For each credential, the local and remote versions are compared individually. Only the newer ones are kept.
4. If there was changes, the resulting merged vault is pushed to the remote server.

This strategy ensures that the system maintains the *most up-to-date vault* without requiring any user intervention. Although this approach is simple to implement, it meets some critical scenarios that must be considered.

Critical scenario 1: Concurrent modifications If the same credential is modified from two different devices while at least one of them is offline,

only the version with the most recent timestamp is kept, resulting in losing the older modification. This approach prioritizes simplicity over completeness, and it does not provide ways to merge the concurrent modifications.

Critical scenario 2: Clock desynchronization The synchronization mechanism relies on timestamps. If one device has significantly desynchronized system clock (e.g., due to manual date and time configuration), the comparison may converge to incorrect results, potentially causing outdated data to overwrite the newer one.

Critical scenario 3: Deleted credentials resurrection A credential deleted locally while offline may still exist in the remote vault. If the remote vault is updated later with respect to the local one, such credential may be unintentionally restored during the next synchronization. To prevent this, deletions could be treated as explicit state changes with their own modification timestamps (i.e., while offline, marking a credential as *deleted*, instead of effectively deleting it).

4.2 Back-end

The **back-end** was implemented to support **user authentication** and **data persistence**, enabling key features such as *cross-device synchronization* and*email functionalities*.

From now on, when a mail sending is mentioned, it means that the back-end is sending an email to the configured *fake SMTP server*, used for debug and testing purposes.

A high-level overview of the system architecture is shown in Figure 14.

The back-end exposes several *endpoints* responsible for managing all the operations related to user authentication and account lifecycle. These endpoints can be grouped into four blocks:

- **User registration**
- **User session**
- **User's password change**

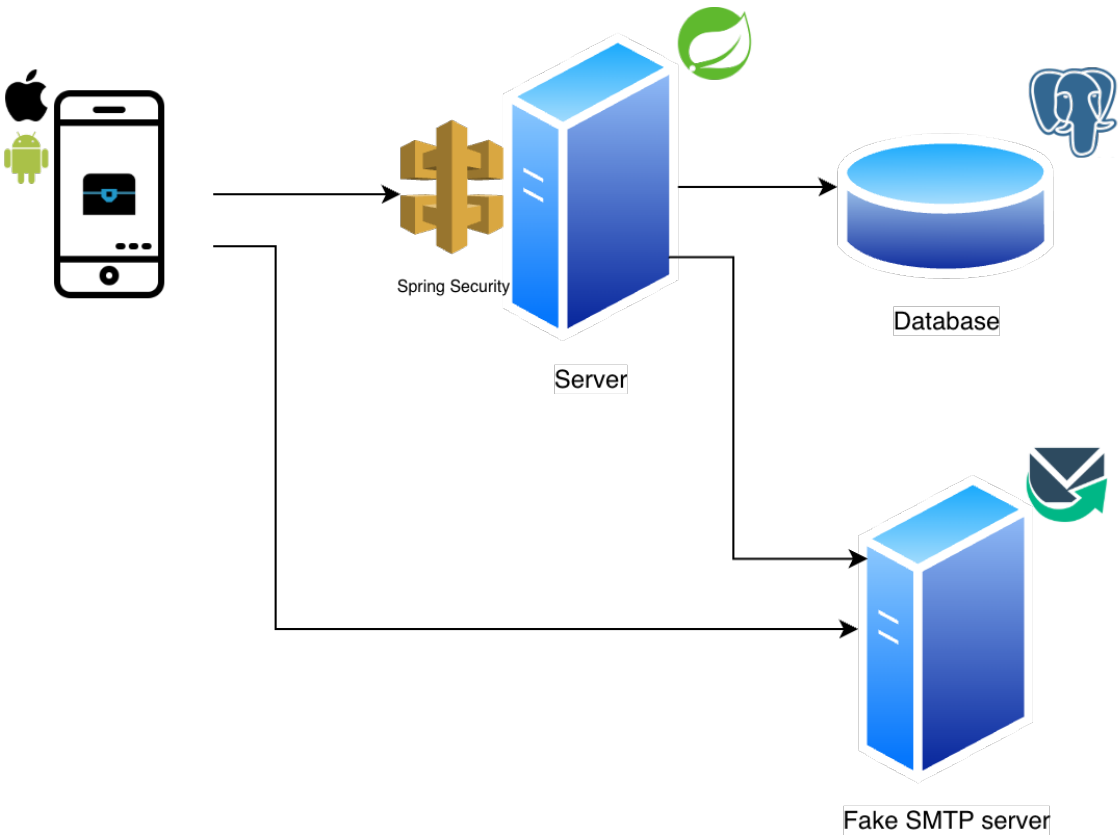


Figure 14: Back-end architecture scheme

- **Account deletion**

In order to guarantee **user's data persistence** (Section 4.2.5), the back-end also exposes an endpoint used to handle valut *retrieval* and *modification*.

4.2.1 User Registration

Before being able to log into the system, the user must first **create an account**. The endpoint responsible for user registration request is **/signup**, which requires the following parameters:

- **Email address**: used to uniquely identify the user inside the system, and to enable a *two-way factor* to confirm user operations (i.e., account creation confirmation, password reset and password modification).
- **Password**: used by the user to authenticate during the login process.

Once the request is successfully processed, the server sends a *confirmation email* to the provided email address.

The email contains a **deep-link** that the user must open in order to activate the account. Opening the link triggers the application to start and automatically send a request to the **/activate** endpoint: if the operation is successful, the user's status is updated consequently and the user can then log into the system using the previously provided credentials.

4.2.2 User session

To log into the system, the user sends a request to the **/login** endpoint, providing the same parameters used during registration (i.e., *email address* and the *password*). If the credentials are correct, releases a pair of tokens:

- **Access token**
- **Refresh token**

The purpose of this mechanism is to provide a *stateless* and secure user session, allowing authenticated access and minimizing the potential exposure of sensitive credentials to third-party.

Access token The **access token** is a short-life JWT used to authenticate the user when accessing protected endpoints. In this project, it is attached by the mobile application to each authenticated request through the **Authorization** HTTP header, using the *Bearer* scheme.

The issued access tokens contain:

- The user's unique identifier
- An expiration timestamp

Its short validity period (in this project set to *1 hour*) reduces the impact of token leakage.

Once expired, it can no longer be used to access protected resources and must be renewed through the *refresh mechanism*.

Refresh token The **refresh token** is a long-life JWT used exclusively to obtain a new access token when the previous one expires. It is never used to directly access protected endpoints.

When the client detects that the access token has expired (when a request gives the *403 HTTP error*), it sends a request to the dedicated endpoint (**/refresh**), providing the refresh token. If the refresh token is still valid, the back-end issues a new access token.

When the refresh token expires, the user must log in again providing their credentials, obtaining a new pair of tokens.

4.2.3 User's Password Change

The back-end exposes some dedicated endpoints to handle the user's **password change** process, in case the user forgets their password or intentionally decide to change it.

Authenticated request (intentional change) When the user is already authenticated, it is possible to send an authenticated request to the **/password** endpoint. When the back-end receives the request, it sends an email to the user containing a *deep-link*, that encapsulates a **request identifier** that uniquely represents the authenticated password change request.

When the user opens the deep-link, the application is launched and automatically shows a page asking for:

- The **current password**
- The **new password**, also repeated for confirmation

Once the form is submitted, the application sends a **PATCH** request to the same **/password** endpoint, providing:

- The request identifier
- The current password
- The new password

If the operations is processed correctly (i.e., the request identifier is valid and not expired, and the current password matches the current one), the back-end updates the user's password. From this moment, the user can authenticate using the new password.

Unauthenticated request (password forgotten) In the case the user forgets their password, it is still possible to reset it by sending a request to the **/password/forgot** endpoint. This requires the user to provide the **log-in email** associated to their account.

By receiving the request, the back-end sends an email to the user to allow them to perform a password reset by opening the *deep-link*, containing the **request identifier**.

The application is launched and a form is displayed, asking the user to prompt the new password twice.

When confirmed, the application sends a **PATCH** request to the **/password/forgot** endpoint that, using the same flow as for the authenticated request, causes the back-end to update the user's password.

Security considerations The request identifier encapsulated in the deep-link has a limited validity in time and can be used only once. This prevents replay attacks and unauthorized password changes in case the link is intercepted by unwanted third-party. For the unauthenticated request, this time is significantly shorter than for the authenticated one. This is justified by the fact that an unwanted malicious user successfully intercepting the deep-link would be able to reset the password with no knowledge of the previous password.

Moreover, this mitigates the risk of **forging** an unauthenticated password reset request.

4.2.4 Account Deletion

If, for any reason, the user decides to permanently delete their account and the associated vault (i.e., all stored credentials), they can initiate the procedure by sending an authenticated **PUT** request to the **/delete** endpoint.

When the request is received, the back-end creates and associates a **delete request** to the user's account and sends a confirmation email containing, embedded in a *deep-link*, the **delete request identifier** that uniquely identifies the deletion operation.

When the user opens the deep-link, the application launches and automatically sends a **DELETE** request to the same endpoint that, if processed correctly, causes the complete deletion of the **user's data** and **login credentials** inside the database.

This operation is *destructive*, and cannot be reverted.

Security considerations The delete request has a validity of 5 minutes, which are considered sufficient for a legitimate user to confirm the operation while significantly reducing the time-window for a malicious third-party in case of email interception attack.

4.2.5 Data Persistence

The primary objective of the back-end is enabling **data persistence** and **cross-device synchronization**. For this purpose, the back-end exposes the crucial endpoint **/vault**, supporting **GET** and **PUT** methods.

Since both the operations involve user related data, authentication is mandatory.

Vault retrieval By invoking the **GET** method, the client retrieves the user's **vault**. The vault is always stored and retrieved **encrypted**, so the back-end itself has no knowledge of its content, and cannot perform any partial manipulation to it.

The only information associated to the vault is the **last modification timestamp**, that is used exclusively for synchronization purposes.

Vault update To upload a new vault version of the vault, the client calls the method **PUT**, providing:

- The **encrypted vault**: a serialized string containing the user's

credentials.

- The **modification timestamp**: representing the time of the last local update performed by the client.

If the provided timestamp is *newer* than the one associated with the currently stored vault, the least is replaced by the one provided by the user. Otherwise, the update request is rejected, preventing any outdated data from overwriting a more recent vault.

Consistency management Since the vault is fully encrypted end-to-end, the back-end is unable to apply any merge strategies, as it is unaware of the partial content (i.e., the single user's credentials).

As a consequence, in scenarios where the same user modifies the vault on multiple devices while offline, conflicts are resolved keeping only the most updated version using the timestamp as comparison method.

I chose this design to prioritize simplicity and confidentiality over solid consistency: this may result in the loss of mid-time updates in concurrent modification scenarios, which are situation I have considered unlikely for the behaviour of an average user.

I developed the back-end following the **Controller-Service-Repository** principle:

- **Controller**: it is where endpoints are defined. No logic, except if related to request dispatching, is performed. A controller delegates *services* to perform the requested operation.
- **Service**: it is the actor that performs logic operations and queries *repositories* to retrieve data.
- **Repository**: it is the bridge that allows for data retrieval.

The entire back-end is deployed inside a **Docker container**, ensuring environment isolation.

4.3 CI/CD

The final step in the lifecycle of a mobile application consists in *building* and *delivering* it to the distribution stores, being them Google's *Play Store* or on Apple's *App Store*.

The upload requires an initial setup, which mainly consists in *creating the application page* inside the store and specifying which developers are authorized to upload new versions. Once the configuration is completed, the deployment process can begin.

Uploading a new version corresponds to releasing an *upgrade* of the application's latest version. This operation can be done either **manually**, directly by the developer, or **automatically**, by adopting a **CI/CD** architecture that, exploiting a pipeline system, automatically does all the steps that are normally done by the developer. The latter approach grants an automated and standardized way of delivery the application to the final user that naturally overcomes possible human errors, ensuring that no mistakes are made.

Moreover, additional steps can be configured to be performed inside the pipeline to increase the final product's quality. In this project, a **code quality checker** was implemented and configured to run before the entire deployment process can start, in order to improve the overall performance and to avoid possible security issues.

Typically, a CI/CD infrastructure is hosted on a *dedicated system*, separated from the developer's environment, which could be located inside the same network or as a **cloud-based service**. However, since this is an individual project, I set up the entire deployment process to execute inside the same machine used for development of the application.

The scheme of the entire building and deployment process is represented in Figure 15.

A crucial component of the entire process is hosting the application project in a **version control repository**. For this project, I used **GitHub** for repository management: it is a widely adopted platform for hosting and managing software projects.

GitHub was primarily adopted to have an online backup of the project

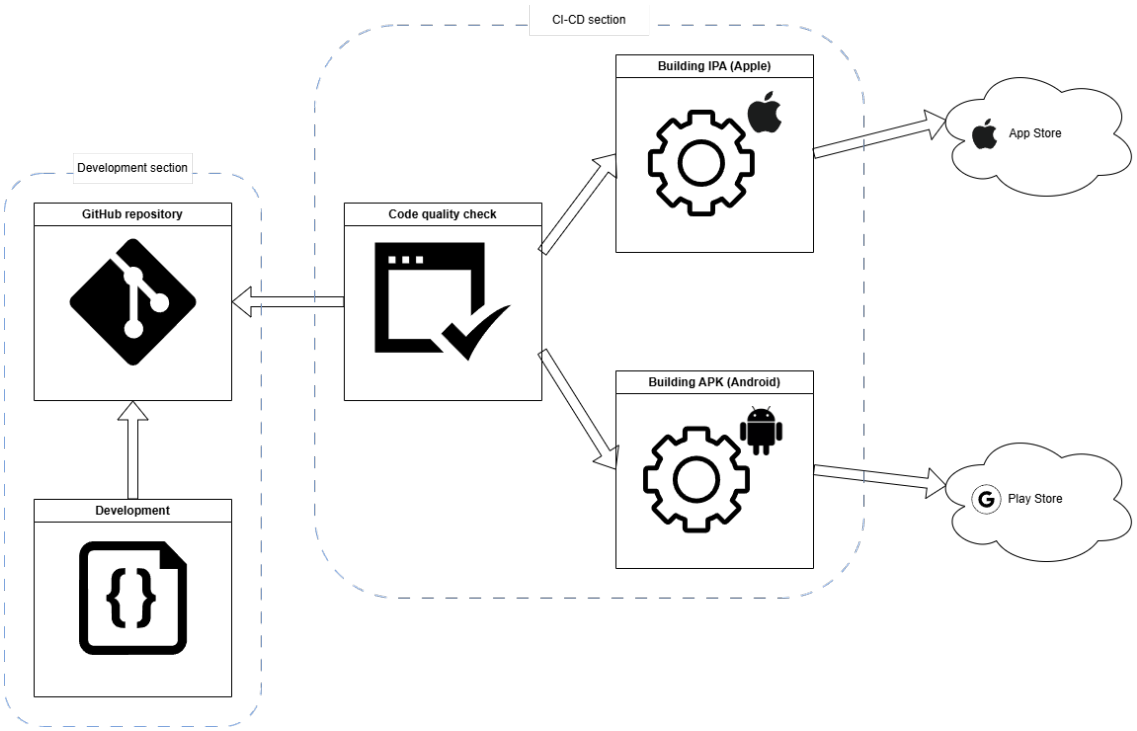


Figure 15: Application build and deployment lifecycle with CI/CD pipeline

and to allow me to work from different locations, but it also represented a crucial component of the CI/CD pipeline.

The repository was configured to include a section dedicated to application's releases. When I want to publish a new version of the application, the code is pushed to a specific repository's branch, referred to as *release branch*.

The automation software (i.e., Jenkins), periodically scans the repository for new project versions. When a new version is detected, the **pipeline** execution is triggered automatically.

4.3.1 Pipeline Configuration

The whole pipeline process relies on Jenkins, and is composed of five steps that are executed in cascade:

1. **Project checkout**: the release version of the project is retrieved from the remote repository by cloning the last release branch.
2. **Code quality analysis**: a static code quality analysis is performed on the source code in order to detect potential *code smells*,

such as duplicated code fragments, security vulnerabilities, and sections with poor maintainability.

3. **Project dependencies download**: all the project's dependencies (i.e., React Native packages and eventual native modules) are downloaded and installed on the project, in order to prepare the project for the building phase
4. **Application packages build**: the project is compiled to produce the final *application packages* for both Android and iOS platforms
5. **Deployment on stores**: the generated application packages are uploaded to the corresponding stores, making the new version available for the download on the respective platforms

The entire pipeline execution is fully automated and does not require any user interaction. Jenkins is responsible for handling possible failures, interrupting the deployment process in case any stage does not complete successfully.

4.3.2 Project Checkout

The **project checkout** phase can be considered the "step zero" of the pipeline, as it is responsible both for *new release detection*, which is the pipeline trigger, and for the *project retrieval*.

I configured the automation system to access resources on a *private remote repository* through a **fine-grained token**, which is an application-specific credential that enables authenticated access and repository resources retrieval.

There are two main configuration strategies to detect a new release:

- **Remote repository polling**: the automation system periodically check the release branch of the remote repository to find new releases and trigger the pipeline accordingly.
- **WebHook trigger**: the remote repository is configured to notify the automation system whenever a new release is pushed on the release branch, automatically starting the execution of the pipeline.

Although the WebHook approach is generally more efficient in terms of resources usage and responsiveness, it is more complex to configure with respect to the repository polling, and requires additional services to be enabled on both the repository and the automation system.

For this reason, I decided to implement the project checkout phase using a *periodic polling strategy*. Once a day, during night time, the system scans the release branch of the remote repository and, if a new version is detected, the project is downloaded locally and the execution of the pipeline is triggered.

4.3.3 Code Quality Analysis

The second phase of the pipeline consists in performing a static code quality analysis, aimed at measuring the overall *maintainability*, *reliability* and *security* of the source code.

The system resposible for this task is **SonarQube** (in this project, hosted inside a dedicated container), which must be up and running before the pipeline execution.

The automation system (i.e., Jenkins) was configured to communicate with the SonarQube endpoint. An **application-specific token** is required for the authentication to the SonarQube server. Moreover, a **scanner** was configured on the automation system's side: it is the component responsible of gathering the source code to be analyzed and then sending it to the server.

In addition, two configuration parameters was provided:

- **Project key**: an unique identifier representing the project to be analyzed by SonarQube
- **Source folders**: a list of the directories containing the portions of source code whose quality must be measured

The automation system triggers the scanner to analyze the specified folders, then the project metadata is sent to the SonarQube server, which will measure the code quality. Several quality metrics are evaluated, including:

- *Code smells* and maintainability issues
- Potential *security vulnerabilities*
- *Bugs* and reliability
- Code duplication and section complex for a human to be interpreted

Once the analysis is completed, SonarQube returns a *quality report* to the automation system.

It is possible to configure a **quality gate**, meaning that the pipeline proceed only if certain predefined thresholds are met. I configured the pipeline to stop if critical issues or blocking vulnerabilities are detected.

If all the quality requirements are satisfied, the pipeline proceeds to the next stage.

Additional note Actually, SonarQube perform static code analysis *without* relying on AI or machine learning techniques. The analysis is entirely based on deterministic methods.

AI-based approaches, by their nature, may give *non-deterministic results*, making them less suitable for critical tasks such as code quality and security analysis, where false positives, or worse, false negatives could lead to bad code to be delivered reflected in potential user's security to be compromised.

4.3.4 Project Dependencies Download

During the third step of the pipeline, the automation system retrieves all the project dependencies required for the build process.

This operation is performed by executing the command `npm install` within the project's root directory. To allow the execution, the **Node.js** runtime environment must be correctly installed and configured on the automation system, along with the appropriate package manager version.

Once all the dependencies installation is completed, the pipeline proceeds to the next step. Compared to the overall execution time of the CI/CD pipeline, this phase generally requires a limited amount of time.

4.3.5 Application Packaging

This phase is the most time-consuming step of the entire pipeline, as it involves the execution of the *Android and iOS toolchains*, which are computionally intensive, especially in the case of iOS.

Android application package building To let the system build a valid Android application package ready to be published on the *Google Play Store*, I went through several configuration steps are required.

First, it has been necessary to own a **Google Play Console account** associated to an organization, with the **Google Play Android Developer APIs** enabled. In addition, a **Google Cloud project** and a dedicated **Google Service account** was created to be able to programmatically access the Play Store.

The service account must have the permission explicitly granted by the organization to release the application on the Play Store.

Android applications must be digitally signed. For this purpose, it was necessary to create a **keystore file** from Android Studio, providing the credentials required to sign the application and to verify its integrity and autenticity.

Once all the preliminary configuration steps are completed, I set the pipeline to compile the project through the following commands:

- `npx expo prebuild --platform android --clean`: generates the native Android project starting from the cross-platform React Native codebase by exploiting **Expo**.
- `gradlew bundleRelease`: builds the **Android App Bundle (AAB)** using the **Gradle Wrapper** script.

The resulting application bundle must then be **signed** using the previously generated keystore file. This operation is performed through the

`jarsigner` tool.

After the signing process is completed, the Android application package is ready to be deployed.

iOS application package building Building a valid iOS application package is significantly more complex with respect to the procedure for Android. First, the entire build process must be performed on a **macOS machine**, since Apple enforces the use of its toolchains (i.e., **Xcode** and related SDKs), that are only available on macOS.

The account setup part follows having an Apple Developer account, inside which it required to have an **Application-specific password**, which is a sort of application-specific token.

It is required to own an active **Apple Developer Account**, from which it is necessary to generate an **application-specific password**, which is a sort of application-specific token that allows automation tools to interact with Apple services.

As for Android, the native source code is generated starting from the cross-platform React Native project by leveraging **Expo**, specifying the iOS platform.

Due to the intrinsic complexity of the iOS build and signing process, a *Jenkins plugin* was used to automate the creation of the application package. Several configuration parameters are required. The most relevant ones are:

- **Development Team ID**: a unique identifier associated to the Development Team, that asserts the developer's membership of the user in an organization enrolled in the Apple Developer Program.
- **Build configuration**: specifies how the application is compiled. To upload an application to the store, the configuration must be set to *Release*.
- **Signing mode**: how the digital signature is applied to the application. In this project, it was set to *automatic*, to let Xcode automatically decide the most appropriate configuration.

To allow automatic signing to work correctly, a valid **developer cer-**

tificate must be installed on the build machine and associated with the configured Development Team.

This certificate is used to sign the application and to prove its authenticity during the App Store validation process. The output of the build process is an **iOS application archive** that can be directly uploaded to the App Store.

Compared to Android, iOS compilation requires a significantly higher amount of time and computational resources, mainly due to the heavier SDK and stricter procedures.

In line with that, the packaging pipeline design process for the iOS ecosystem took more time with respect to the one for Android, due to certificates management and toolchain setup inside the macOS machine.

4.3.6 Deployment on Stores

Once both application packages are successfully built, the deployment process can begin.

Android's Play Store To reduce the complexity of the upload process, I decided to adopt a dedicated *Jenkins plugin* to deploy the Android package on the store. The required parameters are:

- **Google Play account credentials**: these credential are contained inside the *Google Service account JSON key file* created during the previous step.
- **AAB file path**: the path of the generated *Android App Bundle* file, which represents the final artifact produced during the build phase.
- **Release track**: the Play Store channel in which the application is published. Since this project is a curricular project, this parameter was set to **internal**, meaning that the application is uploaded to a **test channel** and made visible only to authorized testers.

Apple's App Store To upload the iOS application to Apple's store, **xcrun**, a command-line tool provided by the Xcode toolchain, was used. This tool allows interaction with Apple services directly from the terminal.

The parameters required for the upload process are:

- **IPA file path**: the path of the generated *IOS Application Package* file, which represents the final artifact produced during the build phase.
- **Apple Developer account credentials**: used to authenticate the upload request.

For iOS applications, the **release track** is not specified at upload time. Instead, it is configured inside **App Store Connect**, where the uploaded build can be assigned to *TestFlight* for beta testing or submitted for App Store review and public release.

5 Testing and Evaluation

This section presents the testing and evaluation done to assess the correctness, robustness and security of the system.

The analysis is structured into three parts:

- **Code quality analysis**: performed to evaluate maintainability, reliability and potential security issues through a static code inspection.
- **Functional testing**: verification that all the functional requirements (defined in Section 2.2) are fullfilled by the implemented features.
- *Security considerations*: evaluation of the effectiveness of the adopted security mechanisms and design choices.

5.1 Code Quality

To assess code quality I enrolled **SonarQube**, which is a tool that automatically analyzing several metrics. The overview of the result I obtained for the project is shown in Figure 16.

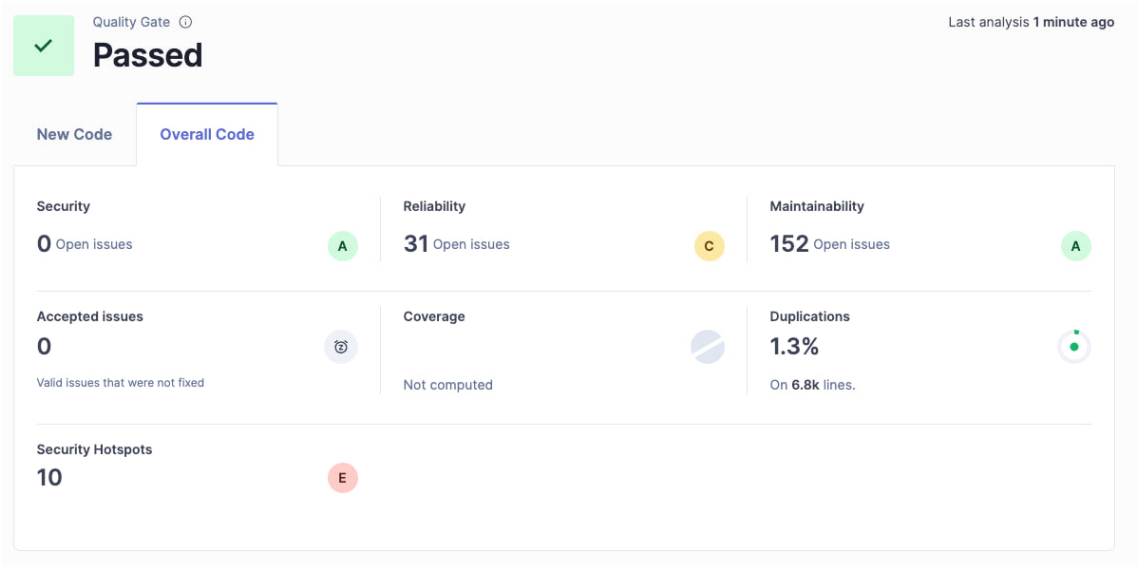


Figure 16: SonarQube static analysis results

The metrics are explained more into details below:

- **Security**: security vulnerabilities (e.g., SQL Injection, XSS).
- **Reliability**: bugs that may cause runtime crashes or unpredictable behaviours.
- **Maintainability**: code smells, so mainly problems related to complexity, duplicated logics, naming, etc.
- **Accepted issues**: user explicitly accepted issues
- **Coverage**: custom automatic code tests (not performed in this project)
- **Duplications**: percentage of duplicated lines of code
- **Security hotspots**: code sections with a potential risk, that require manual intervention (e.g., use of cryptography, credentials management, external input)

As shown by the results, the overall code quality assessment is positive, and the project successfully passes the *Quality Gate* defined in SonarQube.

From a **security** perspective, no vulnerabilities were detected. This result is particularly relevant considering the nature of the system, which manages sensitive data. The absence of reported security issues confirms that no known insecure patterns (e.g., injection or unsafe data handling) are present in the codebase of the project.

Regarding the **reliability**, SonarQube reports a limited number of open issues, classified as potential bugs. More into details, the issues are mostly related to:

- the use of *React hooks* into functions whose return type is kept implicit, making it impossible to assess whether the hook is inside a React context or not
- React node *children* explicitly passed as parameters

These issues do not prevent the application from functioning correctly and did not manifest as runtime errors during the functional testing.

The **maintainability** metric reports several code smells, which are mostly:

- unused declared variable (which I kept in the codebase for possible future implementations)
- use of `var` instead of `let`, which is a more typical Typescript standard
- lines of code into less straightforward formats, which I decided to keep since I was the only project maintainer I found the logics understandable for my way of reasoning
- properties of components not explicitly declarated as constants

While these aspects do not impact the correctness or security of the application, they represent possible improvement for long-term maintainability perspective.

A percentage of **duplicated code** of *1.3%* is low with respect to the total codebase size, meaning a good level of reuse and modularization of the code. This result can be considered acceptable for a project of this size and scope.

Finally, the presence of **security hotspots** indicates code sections that require manual review rather than representing actual vulnerabilities. In this project, such hotspots are mainly related to:

- the use of *regular expressions*, used to validate string patterns (e.g., email, password, etc.), which, despite their easy code readability, are generally considered computationally expensive
- the use of the standard *random* function, implemented in the password generator section to shuffle the extracted characters of the password to be generated

These sections were intentionally implemented ensuring and verifying no drawbacks are met during the functional tests.

As a conclusion, the static analysis confirms that the codebase satisfies acceptable standards in terms of security, reliability, and maintainability, making the project codebase for deployment.

5.2 Functional Testing

In this section, the functional capabilities of the system are analyzed in order to assess the accomplishment of the requirements defined in Section 2.2.

Each test verifies the correct behavior of a specific functionality from the end-user perspective.

- **Sign-up system**: the system correctly allows account creation requesting an *email* and a *password*. After registration, an activation email containing a confirmation link is sent, and opening the link successfully activates the account
- **Login system**: the system allows activated users to log into the system using their *email* and *password*
- **Personal vault**: after authentication, the system provides a personal *vault* associated with the user's account, used to securely store credentials
- **Master password protection**: access to the vault is correctly protected by a *master password*, which is required to decrypt and access stored credentials
- **Biometric vault unlocking**: on supported devices, the system allows vault unlocking through *biometric authentication* as an alternative to entering the master password
- **Vault locking**: the system correctly locks the vault when the configured timeout expires or when explicitly requested by the user
- **Credentials management**: the system allows the user to *create*, *read*, *update*, and *delete* credentials. Each credential includes a *name*, *username*, *password*, and the *associated website*
- **Data encryption**: whenever an operation involving the vault's content is performed, the system is able to *decrypt* the vault, apply the requested operation, and then *re-encrypt* it before storage or synchronization using the *master password*

- **Clipboard integration**: the system allows the user to copy the requested credential field to the system clipboard
- **Toast messages**: when relevant operations are performed, the system shows *toast messages* as feedback for the user
- **Password generator**: the system provides a dedicated section for *password generation*, in which the user can specify constraints and a new password can be *generated*, *copied* to clipboard or directly used to create a new credential
- **Account password change**: the system allows authenticated users to change their account password, requiring the previous one as verification; the user must use the confirmation link that is sent to the provided email in order to confirm the password change
- **Account password reset**: the system allows users that forgot their password to request a reset procedure; the user must use the confirmation link that is sent to the provided email in order to confirm the password reset
- **Master password change**: the system allows logged-in users to change their master password, requiring the previous one as verification
- **Account deletion**: the system allows users to permanently delete their account when requested; the user must use the confirmation link that is sent to the provided email in order to confirm the account deletion
- **Offline support**: the system is able to perform all the essential credential-related operations for a previously authenticated user; after reconnection, it is able to synchronize local data with the remote server
- **Multi-language support**: the system adapts its interface language according to the device system language settings, supporting *Italian* and *English*

This list is a cross-check against the defined **functional requirements**. All required functionalities have been successfully tested and verified.

Therefore, the system is considered to be correctly implemented and fully functional.

5.3 Security Considerations

From a security perspective, which is a fundamental aspect of a credential manager, several key considerations can be highlighted.

User authentication User authentication is implemented using a classical *email and password* login scheme. Passwords are never stored or transmitted as plain text: instead, they are processed using a one-way hashing function, preventing exposing account data even in case of a database compromise.

The authentication process is token-based (Section 4.2.2), ensuring that credentials are exchanged only during the login phase and never reused for subsequent authenticated requests, reducing the exposure of critical data.

Vault encryption All user credentials are stored inside a personal vault, which is **serialized** and **encrypted locally** using a *master password* known only to the user. The master password is never transmitted to the back-end, but is securely stored locally using a dedicated Expo module (SecureStore) to enable *biometric authentication*.

Under the assumption that SecureStore is secure, the master password retrieval cannot be forced in any way, and neither the user's credentials.

Encryption and decryption operations are handled by a React Native module (react-native-aes-gcm-crypto).

End-to-end encryption I developed the system adopting an **end-to-end encryption** model. The back-end only stores and retrieves encrypted vaults and has no knowledge of their content, structure, or individual credentials.

No information related to the master password or decrypted data is ever exposed to the server. As a consequence, even a fully compromised

back-end would not allow third-party malicious users to access user credentials.

Attack surface Given the adopted security architecture, the only possible attack to obtain vault's content is a *brute-force attack* against the encrypted vault, requiring the attacker to guess the master password user for the encryption.

A brute-force attack is computationally expensive and becomes impractical if the user selects a sufficiently strong master password (which is required by default).

Additionally, since both the encryption and decryption processese are performed entirely on the client side, no online oracle or timing-based attack can be exploited through the back-end.

This design choice prioritizes confidentiality over recoverability: if the master password is lost, the vault content cannot be recovered in any way, including server-side recovery procedures.

6 Limitations and Possible Improvements

Although the initial project's requirements are fully met, there are some limitations to be taken into account and consequently there is room for improvements.

The considerations are divided into the relative sections:

- **Front-end**
- **Back-end**
- **CI/CD**

6.1 Front-end considerations

Although the overall testing results are positive, I considered several potential improvements to make the front-end (i.e., the mobile application) more suitable for a real-world distribution scenario.

UI/UX improvements I mainly focused on the functional and architectural aspects of the system, with less emphasis on the refinement of the user interface.

In general, the adopted color palette and layout aim at providing a coherent and readable design, but I did not conduct formal study on the psychological impact of visual elements on the consequent sense of security and trust, which is a crucial aspect for credential management applications.

Moreover, the UX was not object of tailored study cases, but the result of my personal opinion about the hypotetical average user's opinions.

Multi-factor authentication The current authentication model relies on a classical email-password scheme.

An additional layer of security could be introduced by implementing **multi-factor authentication (MFA)** for account login operations, such as one-time passwords (OTP) sent via email. This would reduce the risk of unwanted third-party users authentications.

Implementing multi-factor authentication would require an extension of the back-end authentication logics.

AutoFill functionality AutoFill is a *native* operating system feature that enables automatic form filling through a user-specified credential provider.

Although this project would have represented a valid use case for implementing AutoFill support, no stable and officially supported cross-platform implementation is currently available for React Native.

As a first try, I searched for a way to implement a custom Expo plugin to translate cross-platform code into a platform-specific implementation. The high interaction with native APIs made this initiative out of reach in terms of time of developing complexity.

However, in a real case scenario, this feature would be worth investigating a way of implementing, either through a cross-platform plugin or as a native extension of the system.

Additional credential typologies As a thesis project, i opted for the adoption of single, generic credential format, mainly for applications or websites login data.

Future improvements could include support for multiple credential typologies, such as credit cards, secure notes, or identity documents. This would require extending both the front-end data models and the vault management logic.

Extended credential properties Each credential is currently defined by a limited set of fields (credential name, username, password and site-of-use), sufficient for basic authentication use cases.

An extended implementation could allow customizable or optional properties, such as tags, expiration dates, security notes or password strength indicators.

6.2 Back-end considerations

Regarding the back-end, being mainly designed as a support layer for authentication and data synchronization, I did not find critical limitations in its current scope. However, I considered some possible extension that would further improve the system.

Multi-factor authentication support If the system implemented *multi-factor authentication* as an extended way of authenticating a user, depending on the available MFA mechanisms, multiple authentication factors would need to be associated with the same user account.

Additionally, sensitive operations such as password change, password reset, or account deletion could exploit these additional factors verify the request and reduce the risk of unauthorized actions.

Session and token management improvements The current token-based session model is sufficient for the current system configuration.

A possible improvement to this scheme could be introduced by implementing a *token revocation mechanism* (remote-side session invalidating mechanism), os associating device-bound refresh tokens.

6.3 CI/CD considerations

Regarding the deployment section, two main structural improvements can be considered.

Release channels structuring Thanks to the availability of both a Google Play Console account and an Apple Developer Program account, I was possible to configure the automated deployment on *test channels* for both platforms.

In real-world scenarios, the deployment could be dispatched to different release channels, such as *internal testing*, *closed test* and *production release*, depending on the source branch from which the code is pushed to.

Such configuration would enable a *set-and-forget* deployment model, where, after the initial store page and pipeline setup, the only responsibility of the developers would be pushing code to the appropriate branch in the remote repository, delegating the entire release process to the CI/CD chain.

WebHook-triggered pipeline execution Currently, the pipeline is started when a change in the specified branch is detected via a polling system. This approach is simple to implement and requires minimal configuration on the automation tool side.

However, a significant amount of unnecessary computational overhead is introduced just to continuously check for codebase changes, even when no updates are expected.

Implementing a **WebHook**-based triggering mechanism would significantly reduce the amount of wasted resources, allowing builds to start immediately after code is pushed to the repository.

7 Conclusions

As a final overall consideration, the project reached a level of completeness that, considering the amount of objectives defined by the thesis program, allowed me to touch almost all the crucial phases of a full-stack development lifecycle in a business-oriented project.

My primary interest was focused on *mobile application development*, with particular attention to the *cross-platform approach*. This paradigm is increasingly adopted in real-world projects due to reduced time-to-market and easier maintenance processes. During the development, I was able to observe that cross-platform solutions do not fit every possible scenario, especially when strong interaction with native APIs or top-tier performance requirements are involved, but they represent an effective and sustainable choice for the vast majority of applications, since it significantly reduces development and maintenance effort while avoiding the need for multiple specialized teams.

Although most of my effort was put on front-end development, I also had the opportunity to design and implement a dedicated and fully functional back-end. This experience allowed me not only to understand how a remote server provides its core functionalities, but also to gain a wider understanding on data flows and interactions between front-end and back-end components from both the perspectives, strengthening my skills on both sides.

To complete the full lifecycle of the system, I also worked on setting up an automated deployment pipeline aimed at distributing the mobile application on both Google Play Store and Apple App Store. In particular, I found the iOS deployment process the most challenging due to strict policies, certificate management, and signing requirements. Although the deployment was pointed to a test release scope, this phase represented the final and essential step of the development process: delivering a product to the final users.

The thesis program lasted six months; however, the related project was completed in less than five months of full-time work. It is worth noting that development was not the only activity I did during this period. Approximately one third of the total time was dedicated to studying and understanding the concepts, frameworks, and technologies required

for the project development.

On the front-end side, a significant amount of time was spent understanding the full potential of *Redux* regarding application state management and automated querying mechanism. On the back-end side, the most demanding task was the implementation of the security layer using *Spring Security*. The CI/CD pipeline configuration mainly required a trial-and-error approach, especially due to dependency compatibility, versioning, and certificate-related issues.

Despite the immense and valuable support I received by my company supervisors, who were always available to clarify doubts and discuss technical decisions, the system was developed in total autonomy. This autonomy allowed me not only to learn how to design and implement solutions, but also *how not to*. In my experience, most of the times the substantial portion of technical growth comes from exploration, problem solving, and learning how *not* to build software, which leads to a deeper and more conscious understanding of more correct development practices.

Overall, this project has been for me a complete and highly formative experience. Thanks to the well-structured thesis program and the continuous support of my company supervisors, I could develop a solid competence and a wide spectrum of skills. I strongly believe that, in general, this experience represents an excellent starting point for a professional career as a mobile application developer in complex and high-level development environments.

Acknowledgements

I would like to thank Professor Guido Albertengo for being my thesis supervisor.

Special thanks go to Federico Arvat and Gabriele Onida for the valuable support they provided and for the knowledge they shared with me throughout my thesis program.

I am also grateful to the "Mobile Team" for welcoming me at Lutech during the six months I spent there.

Bibliography

[1] Bitwarden Inc. *Bitwarden Password Manager*. URL: https://bitwarden.com/.

[2] AgileBits Inc. *1Password*. URL: https://1password.com/.

[3] Google. *Google Password Manager*. URL: https://passwords.google.com/.

[4] IBM. *IBM Simon Personal Communicator*. 1993.

[5] Meta Open Source. *React Native: Learn Once, Write Anywhere*. URL: https://reactnative.dev/.

[6] Expo. *Expo Documentation*. URL: https://docs.expo.dev/.

[7] Redux Toolkit. *Redux Toolkit: The Official, Opinionated, Batteries-Included Toolset for Efficient Redux Development*. URL: https://redux-toolkit.js.org/.

[8] Redux Toolkit. *RTK Query Overview*. URL: https://redux-toolkit.js.org/rtk-query/overview.

[9] Redux Toolkit. *Redux Toolkit*. URL: https://redux-toolkit.js.org.

[10] Expo Documentation. *SecureStore*. URL: https://docs.expo.dev/versions/latest/sdk/securestore/.

[11] Expo Documentation. *Expo Clipboard*. URL: https://docs.expo.dev/versions/latest/sdk/clipboard/.

[12] Expo Documentation. *LocalAuthentication*. URL: https://docs.expo.dev/versions/latest/sdk/local-authentication/.

[13] Expo Documentation. *Expo Crypto*. URL: https://docs.expo.dev/versions/latest/sdk/crypto/.

[14] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard (AES) – Galois/Counter Mode (GCM)*.

[15] i18next. *i18next: The Internationalization Framework for JavaScript*. URL: https://www.i18next.com/.

[16] Docker Inc. *Docker Documentation*. URL: https://docs.docker.com/.

[17] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. URL: https://www.postgresql.org/.

[18] VMware. *Spring Boot*. URL: https://spring.io/projects/spring-boot.

[19] Internet Engineering Task Force (IETF). *JSON Web Token (JWT)*. 2015. URL: https://jwt.io/.

[20] Axllent. *Mailpit: An Email Testing Tool for Developers*. URL: https://mailpit.axllent.org/.

[21] Jenkins. *Jenkins User Documentation*. URL: https://www.jenkins.io/doc/.

[22] SonarSource. *SonarQube Documentation*. URL: https://docs.sonarqube.org/.

[23] Google Developers. *Google Play Console*. URL: https://developer.android.com/distribute/console.

[24] Apple Developer. *App Store Connect*. URL: https://developer.apple.com/app-store-connect/.