

# Walman

## A Virtual Wallet Management System

**Candidate:** David Daniel, Pava  
**Coordinator:** Assoc. Prof. Razvan, Bogdan

## REZUMAT

Walman este o aplicație pentru management-ul unui portofel virtual în care utilizatorul își poate stoca în siguranță parolele, card-uri de fidelitate și de shopping și token-uri pentru autentificare în doi factori. Datele pot fi salvate ca backup pe o platformă cloud de încredere sau pe blockchain. Aplicația a fost creată folosind Flutter, un framework cross platform pentru dezvoltare de aplicații mobile.

Pentru backup-ul pe cloud, aplicația a fost integrată cu Firebase și Firestore, două servicii backend ‘pay as you use’ dezvoltate de Google care oferă o implementare rapidă și sigură a unei aplicații de tip server.

Backup-urile pe blockchain sunt implementate folosind smart contracts pe o rețea Ethereum dezvoltate în limbajul de programare Solidity, care este standardul dezvoltării de aplicații decentralizate în momentul actual.

Management-ul stării aplicației este implementat folosind o arhitectură bazată pe Redux, un șablon care minimizează timpul de dezvoltare, reduce costul menenanței și este ușor de testat.

Aplicația are multe caracteristici de la management de parole până la un portofel pentru crypto monede și poate fi îmbunătățită și dezvoltată într-o aplicație comercială.

# ABSTRACT

Walman is a wallet manager designed to help the user safely store their passwords, shopping and fidelity cards and one time password authentication tokens. The data can be backed up on a reliable cloud platform or on the blockchain. The application was created using Flutter, a cross platform mobile development framework.

For the cloud backup, the application was integrated with Firebase and Firestore, a ‘pay as you use’ backend platform developed by Google which offers a quick and secure implementation for a server application.

The blockchain backups are implemented using smart contracts on an Ethereum network developed using the Solidity programming language, which is the state of the art of decentralized application development at the moment.

The state management architecture of the application is based on Redux, a pattern that minimizes development time, reduces maintenance cost and it’s easy to test.

In the end the application has a lot of different features ranging from password management to a crypto wallet functionality and it can be further developed and improved into a commercial application.

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Context . . . . .	5
1.2	Motivation . . . . .	6
1.3	Similar Products Available on the Market . . . . .	6
<b>2</b>	<b>Technology Stack</b>	<b>8</b>
2.1	Frontend . . . . .	8
2.1.1	Flutter . . . . .	8
2.1.2	Dart . . . . .	9
2.1.3	Code Generation . . . . .	10
2.1.4	State Management . . . . .	10
2.2	Firebase . . . . .	11
2.3	Blockchain . . . . .	12
2.3.1	What is a Blockchain? . . . . .	12
2.3.2	Ethereum . . . . .	13
2.3.3	Smart Contracts . . . . .	14
2.3.4	Solidity . . . . .	14
2.4	Development Environment . . . . .	14
<b>3</b>	<b>Implementation</b>	<b>16</b>
3.1	Use Cases . . . . .	16
3.2	System Architecture . . . . .	18
3.3	Application Flow . . . . .	19
3.4	Application State . . . . .	22
3.5	Application Action . . . . .	25
3.6	Epics . . . . .	26
3.7	Reducers . . . . .	26
3.8	Application Models . . . . .	28
3.9	Password Management . . . . .	30
3.9.1	Password Generator . . . . .	30
3.9.2	Create/Edit a Password . . . . .	31
3.9.3	Delete a Password . . . . .	34
3.10	QR and Barcode Management . . . . .	35
3.11	OTP Authenticator . . . . .	37
3.11.1	TOTP . . . . .	38
3.12	Cryptocurrency Wallet . . . . .	40
3.13	Backup . . . . .	41
3.13.1	Cloud Backup . . . . .	41
3.13.2	Blockchain Backup . . . . .	41
3.14	Security . . . . .	44

<b>4</b>	<b>Tests</b>	<b>46</b>
4.1	Performance Statistics . . . . .	46
<b>5</b>	<b>Conclusions</b>	<b>47</b>
5.1	Limitations . . . . .	47
5.2	Possible Improvements . . . . .	47
	<b>Bibliography</b>	<b>49</b>

# 1 INTRODUCTION

In this project I implemented a wallet manager in the form of a mobile application. The main features are password management, password generation, qr and barcode storage and management, crypto wallet and OTP token management. The user has the option to backup the data either in the cloud or on the blockchain. Once backed up, the data will be available to be restored.

## 1.1 CONTEXT

In the last 30 years, the number of tasks that are digitalized has increased exponentially. The most important part of the security systems of these tasks is user management and authentication. The password is the most widely spread form of user authentication and thus is often the prime target of attackers that want to impersonate someone else.

According to [7], a “*systematic literature review in the area of passwords and passwords security*”, there are many problems with password security and management ranging from weak passwords and password reuse to users writing down passwords or sending them through unsecure channels. Most of these problems according to [7] are solved by using password recommendations. A good solution to most of these problems is a password management tool. A password manager is a piece of software designed for generating and managing passwords, in this way the user can have unique, complex and safely stored password without having to remember them.

Another great method to better secure you accounts is using a two factor authentication (2FA) method. These method vary from security questions, to one time passwords (OTP) sent from the server to the user via email or SMS, to OTPs generated using specialized algorithms such as: HMAC-based One Time Password (HOTP)[8] and Time Based One Time Password (TOTP)[9].

The cryptocurrency market is another area that has seen a considerable development lately. As of May 2022, the market cap of Bitcoin is around 565 billion USD, and the market cap of Ethereum is around 214 billion USD. In the case of Bitcoin, that is more than double of what it was in 2019 (around 211 billion USD), referenced in [1]. Cryptocurrencies also offer secure and long-term storage capabilities thanks to the blockchain technology. Blockchain backups, thanks to the decentralized nature of the blockchain, are very hard to be tempered with. A traditional cloud backup could be lost or inaccessible in more than one situation. The most obvious one is data loss happening as result of a cyber-attack or the company simply going bankrupt. There are also situations in which the company itself can refuse to serve you anymore. They can freeze your account or just refuse to serve an entire country all-together, we have the recent example of companies like Visa and Mastercard refusing to serve Russian citizens as result of political tensions as described in [6] and [3]. All these scenarios cannot happen in a decentralized blockchain system.

Businesses that were traditionally not online like shopping also have inversely digitalized. Nowadays most of the hypermarkets offer fidelity cards. Usually, these cards are built around a unique barcode or qr code. Often, it's hard to manage all your cards, so a digital storage solution to solve this issue would help the end user better manage their cards.

Considered all mentioned above, a user must remember and manage a lot of information in order to interact with the currently available online infrastructure. A tool that could help them manage all this data better is a wallet manager.

## 1.2 MOTIVATION

My personal motivation for creating a wallet manager is the fact that I want to use it myself. Also, I wanted for a long time to explore the state of the art in smart contract development, so this was a great occasion to do so.

I chose to create this project in the form of a mobile application since people tend to have their smartphones with them most of the time, so having a virtual wallet on your mobile device makes sense.

Another factor that motivates me is the fact that currently in the mobile application market there are almost no free and open-source password management applications available. The user needs to *trust* the creators of the application with their data, not knowing how the implementation of the product is made, they have no guarantee that the data is safe.

## 1.3 SIMILAR PRODUCTS AVAILABLE ON THE MARKET

There are a lot of password managers available on the market. In this section we are going to try to make a comparison between some of the most popular options available.

Property	LastPass	RememBear	KeePass	PassMan	KeyBase
Mobile Version	Yes	Yes	No	Yes	Yes
Blockchain Storage	No	No	No	No	Yes
Price	\$3/month	\$6/month	Free	Free	Free
License	Proprietary	Proprietary	GPL-2.0	AGPL-3.0	BSD-3

Table 1.1: A comparison between some of the most popular password managers.

In Table 1.1 we have a comparison between some of the most popular password managers available on the market. First off, we have LastPass[22] and RememBear[27], two very similar password managers, both having a free and a paid plan. Neither of these two is open-source, so the most pressing issue regarding them is the guarantee that your data is safe. Without having the ability to see how your data is managed and stored you cannot be certain that it is secure. Also, these applications do not have blockchain backups.

KeePass[20] is probably the most popular password manager for desktop. It is free and open-source, and the code was analyzed and certified by specialized organizations such as the Open-Source Initiative. The biggest drawback to KeePass is the aged user interface and the missing mobile application counterpart. Nowadays a lot of the situations where a user needs access to their credentials are happening while using smartphones. Also, the features are limited, KeePass doing one thing and doing it well that being password management. There is no cloud or blockchain backups, so the user needs to manage backing up and storing their password database themselves.

Similar with KeePass, PassMan[24] is a free and open-source password manager. They have a mobile version of the application, but blockchain backups are missing. Also, again, PassMan is just a password manager. It does not manage shopping cards or crypto wallets.

Last but not least there is KeyBase[21] which is not technically a password manager. KeyBase is a blockchain, decentralized, social media application. You can store password and secure notes inside the application but from the user experience point of view, KeyBase was never designed to be a password or wallet manager. The reason why it is mentioned, is because KeyBase is implemented on the blockchain, all user data is encrypted and it's free and open-source.



## 2 TECHNOLOGY STACK

In this chapter I am going to describe the technologies used. The application has 3 main components. The first component is the frontend, a mobile application. The second one is the cloud storage backend. The last part are the smart contracts deployed on the blockchain.

### 2.1 FRONTEND

The most important aspects I considered when I chose the technologies used for the frontend was cross-platform capabilities, ease of testing, documentation availability and performance. Therefore, for this project I chose a Flutter stack.

#### 2.1.1 Flutter

Flutter[16] is a mobile application development framework developed by Google in the Dart programming language. It was released in May 2017, and it currently is one of the most popular mobile development frameworks.

According to “An empirical investigation of performance overhead in cross-platform mobile development frameworks”[2], Flutter has one of the better resource management systems when compared with other popular mobile development frameworks.

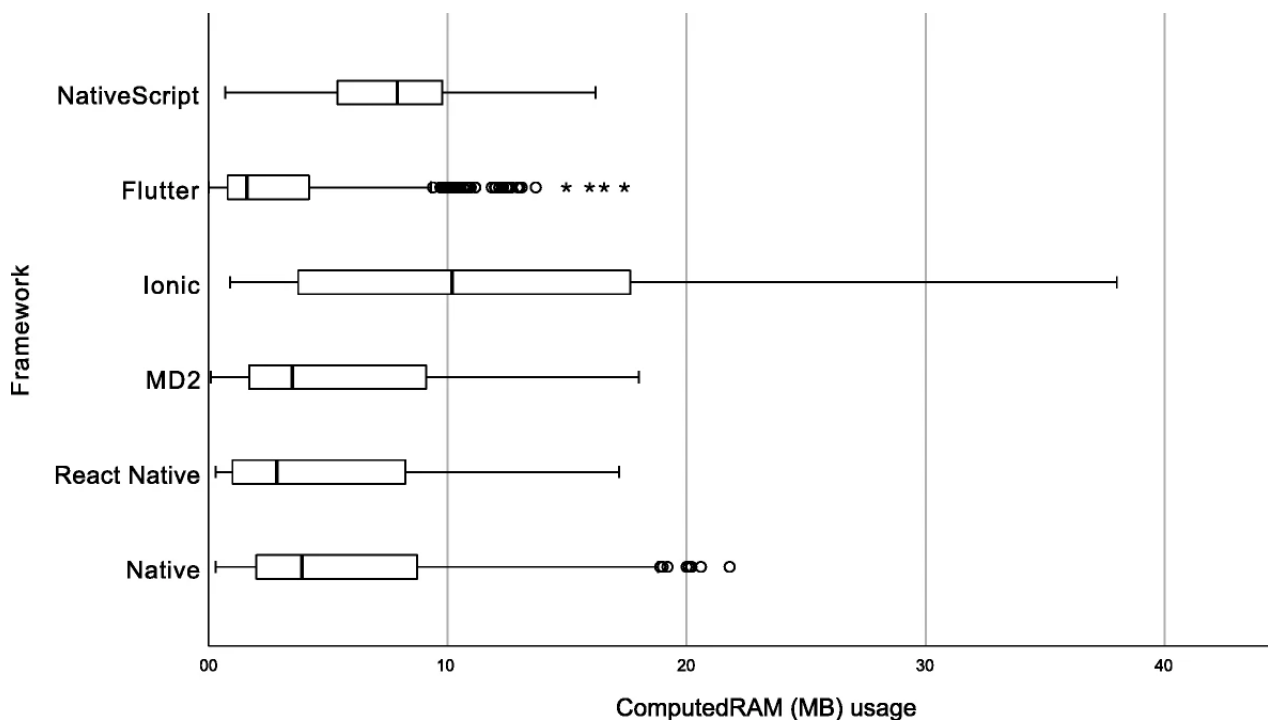


Figure 2.1: Boxplot from [2] of RAM usage across all tests done in [2]

As seen in Figure 2.1, on average Flutter outperforms most of the other frameworks in memory management.

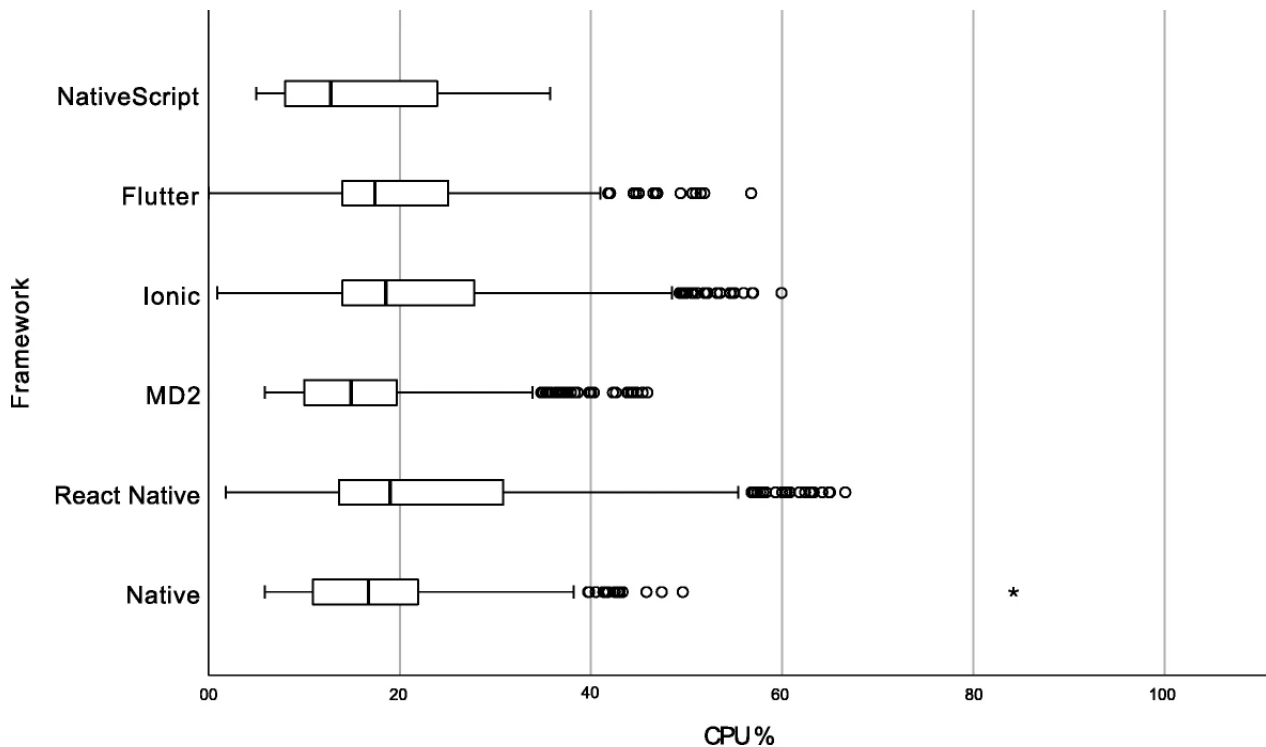


Figure 2.2: Boxplot from [2] of CPU usage across all tests done in [2]

In Figure 2.2 we have a comparison between CPU usage of similar applications implemented in different frameworks in [2], where Flutter achieves a competitive result when compared to the other frameworks.

Another very important feature of Flutter is cross-platform compatibility. A mobile application developed in this framework can be built in native Android and IOS applications with minimal performance loss. There is also Flutter Web for web applications, offering the option to create a browser variant of the application in the future, reusing already developed and tested parts from the mobile application.

Flutter comes with a very rich and detailed documentations, the Flutter Docs[16] and a set of development plugins for the most used IDE and text editors such as Android Studio, IntelliJ or Visual Studio Code. During the development process the code is executed into a runtime environment, allowing almost instant compilation times speeding up the development, and in production, the code is compiled into a native application for performance enhancement.

The User Interface in Flutter is build based on a widget tree, like React. Every User Interface item inherits the Widget class.

### 2.1.2 Dart

Dart[12] is a general purpose programming language developed by Google starting with 2011. It was intended to replace JavaScript and TypeScript for frontend applications, but instead, later, it was used to create the Flutter framework.

Dart is a type safe, C-like programming language. It can be both interpreted by a runtime or compiled. The memory management is handled by a garbage collector similar with Python or JavaScript.

One of the strongest features of Dart is the compiler. It can be compiled in binary code,

JavaScript or mobile native code such as Java and Kotlin for Android and Objective-C and Swift for IOS devices. Dart also performs tree shaking at compile time, discarding unused objects, methods and functions.

### 2.1.3 Code Generation

During the development of the project, I used multiple packages (dart libraries) in the process. One very important package that needs to be mentioned is the *freezed* package[19]. This offers code generation for common model functionalities such as json encoders and decoders, copyWith methods, different constructors and access functions and more.

Classes generated by the package are annotated with the *@freezed* tag. The generated code is stored into files that contain the *‘.freezed.dart’* and *‘.g.dart’* extensions.

### 2.1.4 State Management

One of the most important aspects of frontend application development is state management architecture. There are a lot of different state management patterns available in Flutter[18] such as Provider, BLoC or the simple setState. These state management patterns tell the application when the state has changed and when certain components of the presentation layer (the UI of the application) need to be updated as a result of that.

For this project I used the Redux state management architecture. This pattern is a very popular solution for managing the state of an application, and it is commonly used in web development. There is an implementation for it in Flutter in the packages: flutter\_redux[17], redux[25] and redux\_epics[26].

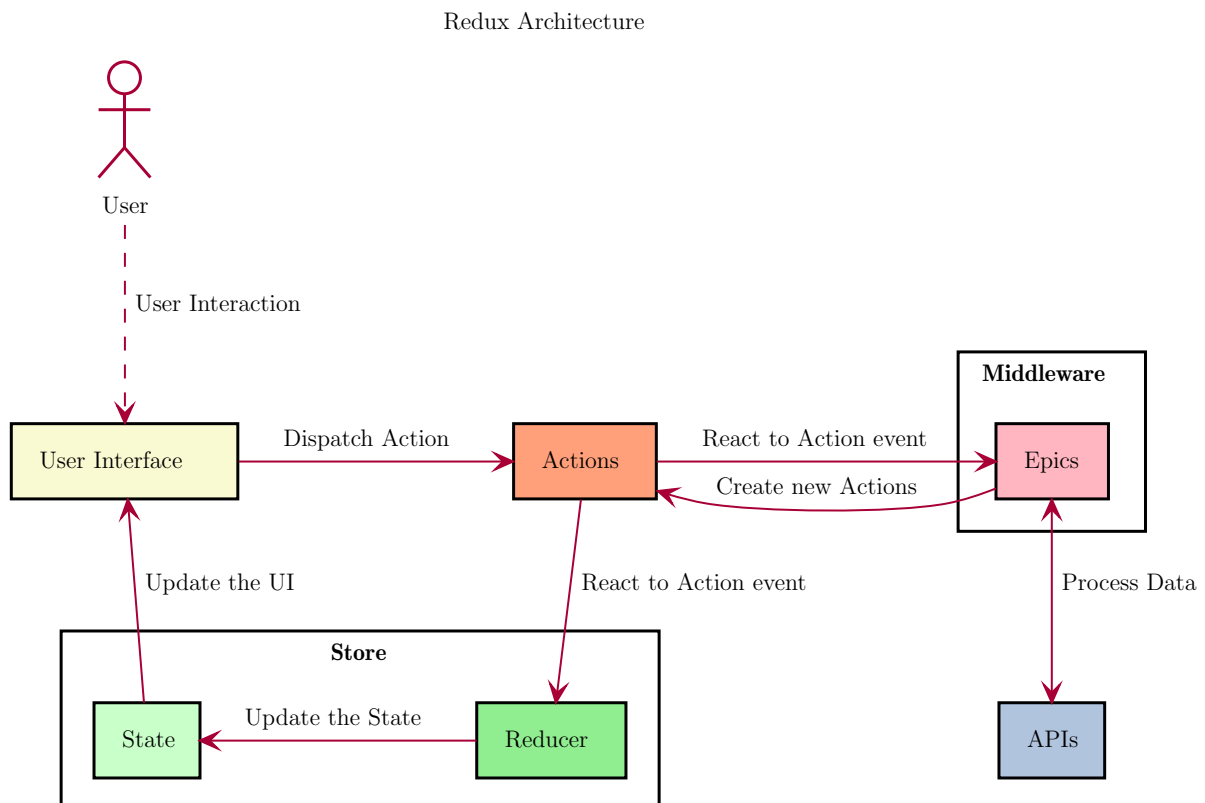


Figure 2.3: Basic structure of a Redux State Management System

In Figure 2.3 we can see the core structure of the redux architecture I used in the project. A state change begins the dispatch of an action. They are usually triggered by the user interface, but in some cases, they can be triggered by an API event. The Epics are a set of listeners which analyze the action stream. When an Epic recognizes an Action, it performs a series of operations which can process the data using the APIs (business logic) or dispatch new actions. Every action is also watched by the Reducer, which listens for Actions and changes the State accordingly. When the State changes, the Widgets (in case of Flutter) that depend on the elements updated in the State, are triggered to be updated.

In order to access specific elements of the state, and not update every widget, every time the state changes. Containers can be used to access a specific part of the state. In order to access the State, the User Interface needs to request it from the Store, in this way, the Store knows when to redraw the object. The widgets that update when the state is modified and have access to the Store are part of the flutter\_redux package[17].

Error handling in redux is made using special stream functions from the RxDart package[28]. RxDart offers an extension to the functional capabilities of dart. In redux the application is represented as a stream of actions. In order to make error handling efficiently, every action sequence spawns a new stream. In case of an exception, the stream will have an exception and we can dispatch a new Action with relevant information about the error. In this way the probability of total application wide runtime exceptions is dramatically lowered.

## 2.2 FIREBASE

In this section I am going to describe the technologies used for the creation of the cloud storage server portion of the application. In order to use the cloud storage, the user needs to create an account. After the account is created, the user needs to have the ability to create backup entries and restore previous backups.

Firebase[15] is an app development platform created by Google, designed as a backend for mobile and web applications. Firebase offers already implemented solutions for user management, artificial intelligence integration and databases. A firebase backend is hosted by Google in a “pay as you use” monetization scheme.

For this project I chose to use the services provided by Firebase to implement the cloud storage part of the project, mainly for ease of deployment, the good integration with Flutter throughout the dedicated packages and reduced costs (so far free).

The user management uses the email and password login service provided by Firebase. Once a user account is created, a database entry is also made, access to this database being restricted to the user base on their unique identifier.

For the storage, Firebase provides two different services. The first one is Real Time Database, which is a NoSQL type database, that resembles a Json type file. There are drawbacks to the Real Time Database such as low fragmentation which increases the data transfer size between the client and the server. A solution to this problem is the second service, Firestore, which is also a NoSQL type database similar with Real Time Database but it has some additional features. Firestore is structured into collections of documents which each have multiple fields of different types. This approach allows the client a more granular access to the data, reducing the size of files transferred between the client and the server and client-side processing.

## 2.3 BLOCKCHAIN

For the Blockchain storage of the application I used the Ethereum smart contract development stack with Solidity. In this chapter I am going to explain the basic logic behind the blockchains, how smart contracts are built into them and the economy between smart contract deployment and usage.

### 2.3.1 What is a Blockchain?

The blockchain is the basic structure that sits at the base of most of the cryptocurrencies. At its core, it is a decentralized, distributed system of data storage and processing.

Block
blockNumber
timestamp
nonce
stateRoot
transactions
difficulty
baseFeePerGas
parentHash
mixHash

Figure 2.4: The structure of a block as described by the Ethereum documentation[14]

The blockchain is made up of a list of blocks connected to each other. Every block contains a set of information about itself and a reference to the previous block, the *parentHash* field in case of Ethereum, in the form of a hash as described in Figure 2.4. The *parentHash* of the current block is the *mixHash* of the previous block.

The *mixHash* is the digest of a hashing algorithm of the entire content of the block. The *mixHash* needs to have the first two bytes 0, in order to be considered mined. This is achieved by incrementing the *nonce* field until the target *mixHash* is reached.

For the hashing algorithm, Ethereum was developed using the Keccak-256 hashing function, which was later standardized as SHA-3 in [4].

The block also has the *timestamp* when it was mined and the *stateRoot* which contains metadata about the current state of the system.

Information about the difficulty of the mining process and the respective price of the process are also stored in the block, in the *difficulty* and *baseFeePerGas* fields.

The *transactions* field contains information about the transactions associated with the block and the optional data exchanged. This is the place where the smart contract logic is deployed and later interacted with.

A successfully mined block is added to the blockchain and then the transaction is confirmed by every subsequent blocks.

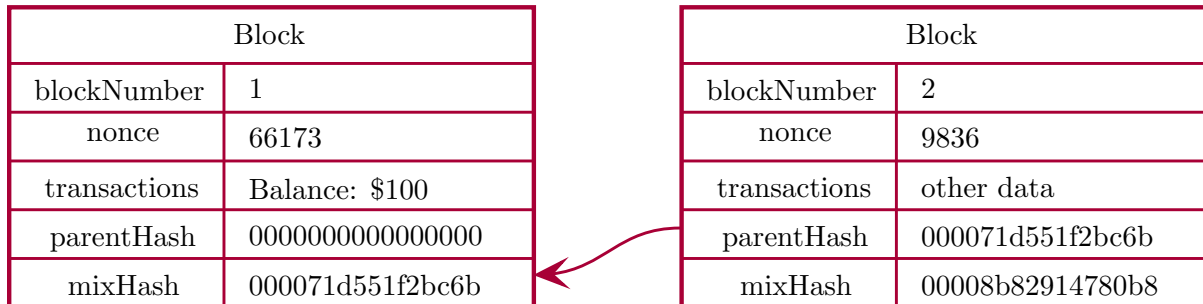


Figure 2.5: High level relationship between two blocks

In Figure 2.5 the relationship between two blocks is represented. As stated before, *block 2* has the mixHash of *block 1* in its parentHash field. In this example, *block 1* being the first block in the blockchain has the null value as parentHash.

If the data in *block 1* was to be changed, it would invalidate the mixHash of *block 1*. The mixHash would have to be mined again, but then it will be different than the parentHash of *block 2*, therefore invalidating the blockchain. This is the first mechanism of defense against malicious data manipulation. There cannot be changes in any previous blocks of the blockchain without invalidating every block starting from the change.

The blocks following a changed block can be mined again in order to re-validate the blockchain. Here is where the second line of defense comes in, distribution. Every chain is stored on multiple nodes, therefore a change in one must be reflected in all of them. This makes data alteration nearly impossible.

### 2.3.2 Ethereum

Ethereum is one of the most popular cryptocurrencies. Other than the popularity and the implicit high amount of resources that is implied, and unlike other popular cryptocurrencies like Bitcoin, Ethereum offers the ability to run code on the blockchain using the Ethereum Virtual Machine (EVM)[14].

Every execution on the Ethereum Virtual Machine requires Gas, which is an amount of Ethereum cryptocurrency units (ETH) proportional with the complexity and the memory requirements of the task. Since the cost of an ETH has raised dramatically over the last 5 years, subdivisions of the currently were created. The two most widely used subdivisions are *wei* ( $10^{-18}$  ETH) and *gwei* ( $10^{-9}$  ETH).

Since the development of a blockchain application implies a lot of testing and prototyping, solutions of simulating the blockchain were created. Ganache is such a software developed in JavaScript, it simulates an Ethereum blockchain on a local machine or on a server. For this project I tinkered with Ganache but later I decided to go one step further and deploy my contracts on an Ethereum Testnet.

Ethereum Testnets are networks similar with the main Ethereum network (mainnet), designed to provide a free of charge development environment identical with the mainnet. Testnets

are blockchains that run in parallel with the mainnet, having the same functionalities and features. There are a lot of Ethereum Testnets like: Ropsten, Kovan or Goerli. In this project I worked with the Rinkeby network. Unlike on the mainnet, ETH is generated constantly on the Testnets and it is distributed to any wallet that requests it via faucets. There are no costs involved with developing a blockchain application on a Testnet. After the development process is complete, in production the application can be easily redeployed on the mainnet.

### 2.3.3 Smart Contracts

The code that can be executed by the Ethereum Virtual Machine must be compiled into a specific binary code. Smart contracts are transactions that contain binary code that can be later executed from subsequent transactions. Gas cost afferent to the execution of the contract is paid by the transaction that wants to execute the contract, not the contract itself. Therefore, in order to create a blockchain application, one of the requirements of such a program is having an attached cryptocurrency wallet. In the case of server applications, this wallet can be a server secret, managed by the administrator, but in the case of client applications, the wallet cannot be incorporated by default into the application, since that would mean that everyone has access to it and is able to act maliciously. This is the reason why this project requires an integrated wallet. An external wallet can also be used, but it would deteriorate the user experience.

Once a smart contract is deployed, it cannot be changed, therefore testing such a contract for vulnerabilities and bugs is essential.

One very important aspect of blockchain applications is gas efficiency. Certain operations consume more gas than others, and storing data also consumes gas. The application should be optimized to leave a minimal data footprint focusing on using non persistent memory.

Accessing data from a smart contract is free of charge. If there is no new data to be stored or there is no code to be executed, accessing the data in a smart contract is cost free.

A very important security aspect of smart contracts and blockchains in general is that data is always visible by anyone who accesses the blockchain and it cannot be modified. Therefore, sensitive data needs to be encrypted.

### 2.3.4 Solidity

Solidity[29] is a dedicated programming language for blockchain applications. It was designed specifically for the creation of smart contracts. This language is an object oriented, compiled language that resembles C++ and JavaScript.

Code written in Solidity is compiled into binary code that can be executed on the Ethereum Virtual Machine and there are specific modifiers for the interaction with the EVM and code optimization. Some of these modifiers are *view* which specifies that the afferent function is forbidden from changing the state and *payable* which allows ETH to be transferred with the function call. There are also more commonly used modifiers like: *override*, *virtual*, *private* or *public*.

## 2.4 DEVELOPMENT ENVIRONMENT

For the creation of this project, I used a wide variety of tools. They range from Integrated Development Environments (IDEs) to task management applications.

For the development of the mobile application, I used IntelliJ and Visual Studio Code with the Flutter specific plugins. I also used Adobe XD for user interface design and plantuml for diagrams. In order to test the application, I used the Android Virtual Machine provided by Android Studio.

For the blockchain part of the application I wrote the smart contracts using Solidity. I created a python compilation script for the smart contracts that is also used in the testing pipeline.

For the cloud portion I used the Firebase console and the specific packages for Flutter.

All the code was managed using the Git software versioning system and it was deployed on Github at <https://github.com/dvpv/walman>. I used the Github Actions feature in order to manage the two test pipelines for the mobile application and blockchain smart contracts.

The smart contracts were deployed on the Rinkeby Testnet using the Remix Ethereum IDE, and in order to access them from the mobile application, the web3dart[31] flutter package was used.

For task management I used Focalboard, a free and open-source alternative to Trello.



## 3 IMPLEMENTATION

### 3.1 USE CASES

The application has six main components: the password, QR and barcode, OTP and crypto-wallet managers and the cloud and blockchain backup solutions.

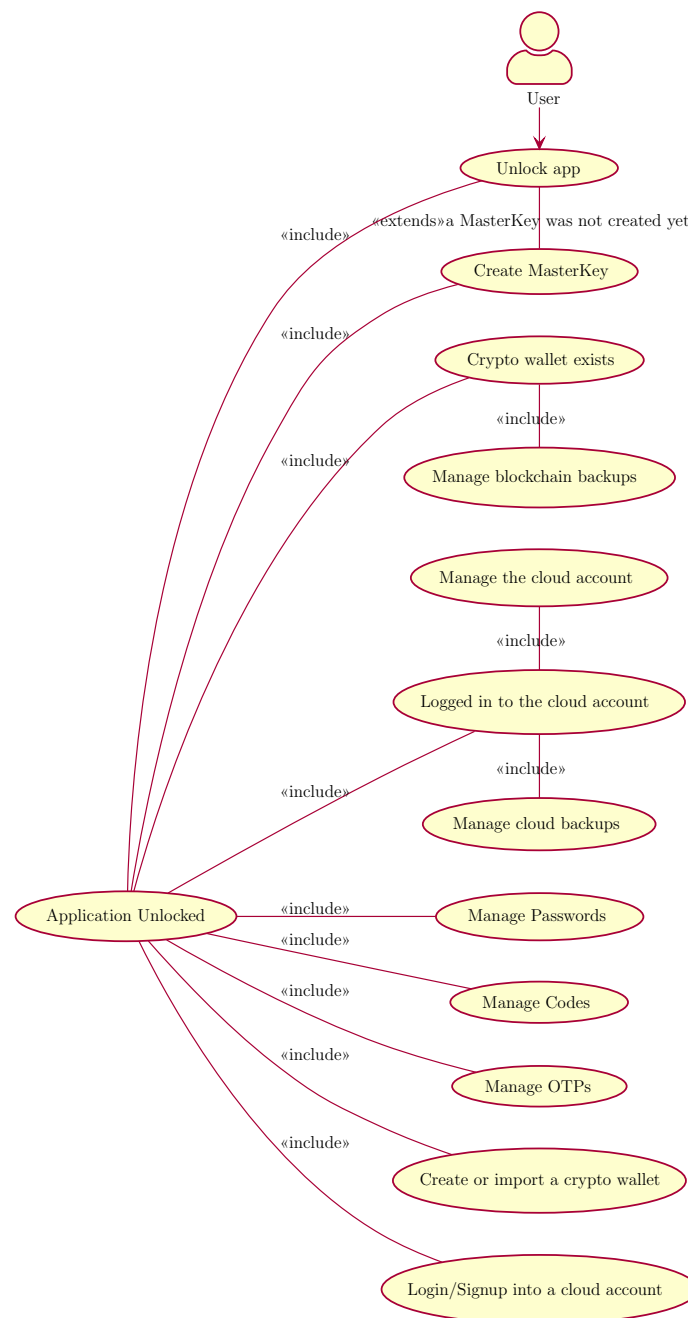


Figure 3.1: Principal use cases of the application

In Figure 3.1 the main use cases of the application are depicted.

The entire application is locked and encrypted using the mater key. If a master key was not set yet, in the case of a first time launch of the application, the user is prompted to create a new master key. After the application is unlocked, the user has access to multiple functionalities and features.

The home page of the application shows in a short format the passwords, codes and wallet balance. The user can navigate to different pages for each of the functionalities.

The passwords page shows the user a list with the passwords that are currently stored into the database. Here the user has the option to add, edit or remove passwords.

The codes page is similar with the passwords page, the user can scan new codes, view existing ones and delete them.

On the OTP page, the user can see the currently stored one-time passwords. The codes are updated every second and a timer representing the remaining time in which the current code is valid is displayed.

On the wallet page, the user can see the current balance of the wallet and the public address. The only transactions that can be made with this wallet are blockchain backup operations.

The sync page shows all available cloud and blockchain backups. The use has the option to restore a backup or create new ones.

In order to create a blockchain backup the user needs to have a crypto wallet. If there is no crypto wallet created, the user has the option to import one by private key or create a new one. For the backup to work, the wallet needs to have enough currency in it in order to pay for the gas price. Restoring a backup requires no gas payment. Crypto wallets are not backed up on blockchain backups and neither they are on cloud backups.

For a cloud backup to be created the user needs to be logged into a cloud account. If the user is not yet logged in, they can create new account, or login into an existing one.

On the settings page, the user can manage the cloud account and view and delete the wallet private key.

There is an app-wide search button which displays results from the entire application. If the user clicks on one of the results, they will be redirected to the respective page in order to view the content.

### 3.2 SYSTEM ARCHITECTURE

The system is composed out of three main components. The communication between these components is made exclusively using the HTTPS protocol.

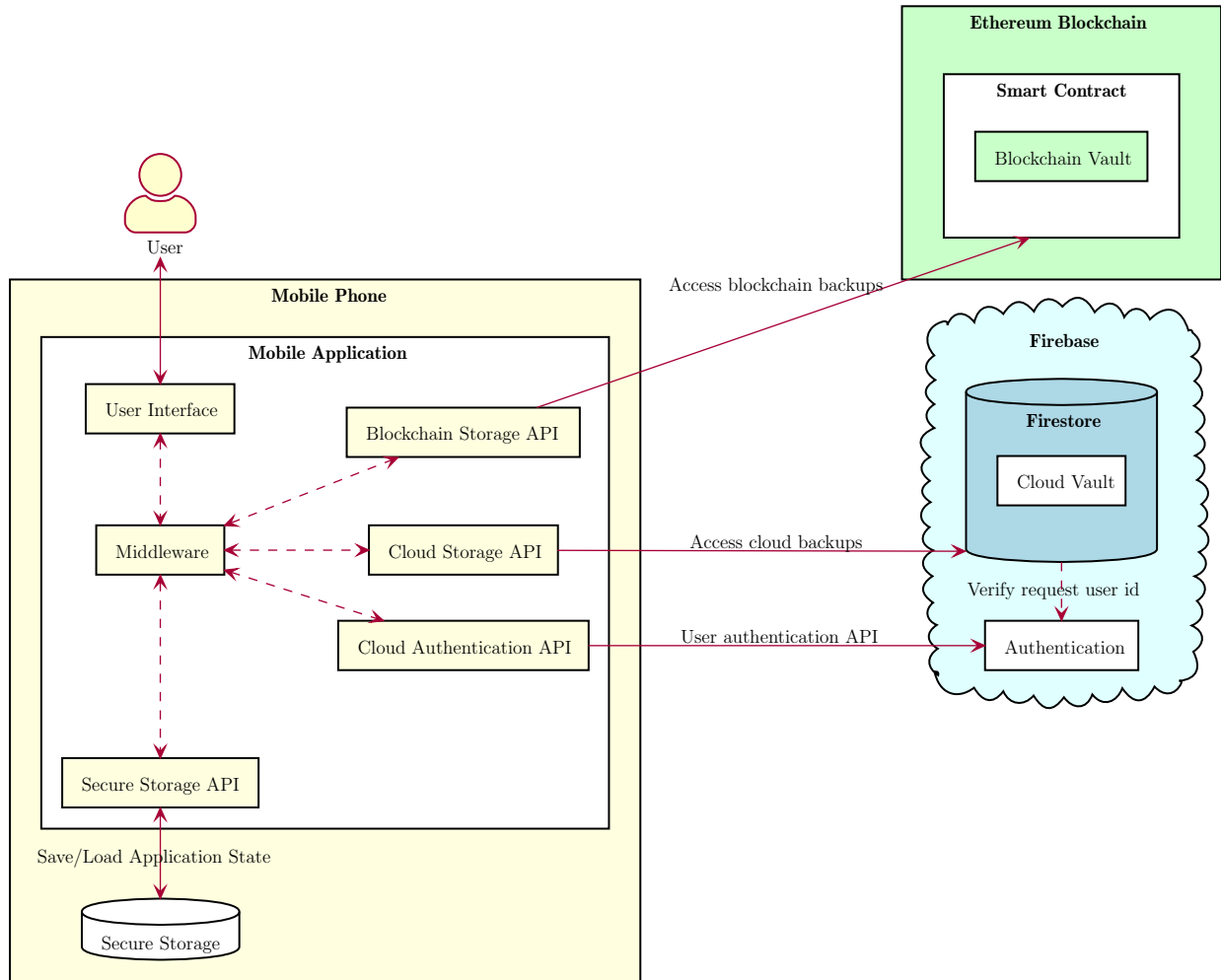


Figure 3.2: High level system architecture

In Figure 3.2 a high-level representation of the system is depicted into a component diagram. The user can directly interact only with the mobile application. The Firebase component is deployed online in the cloud platform provided by google and the smart contract is deployed on the Ethereum blockchain.

The mobile application is the first component, and it is the only component that initiates data exchange in the system. It has three API sub-components for each of the online services it needs to communicate with and one API for interaction with the Secure Storage.

The Secure Storage (or Secure Enclave on IOS) is a special persistent memory zone created to store sensitive information on a mobile device. I used the Secure Storage to save the part of the application state that needs to be recovered at each application start.

The middleware consists of the redux epics defined into the application. The epics are the only part of the mobile application which has access to the external APIs.

The user interface interacts with the middleware by dispatching redux actions. The state of the application is updated by the reducer when it detects specific action events that require

a state update. The user interface code is completely separated from the APIs, the only connection being the middleware.

### 3.3 APPLICATION FLOW

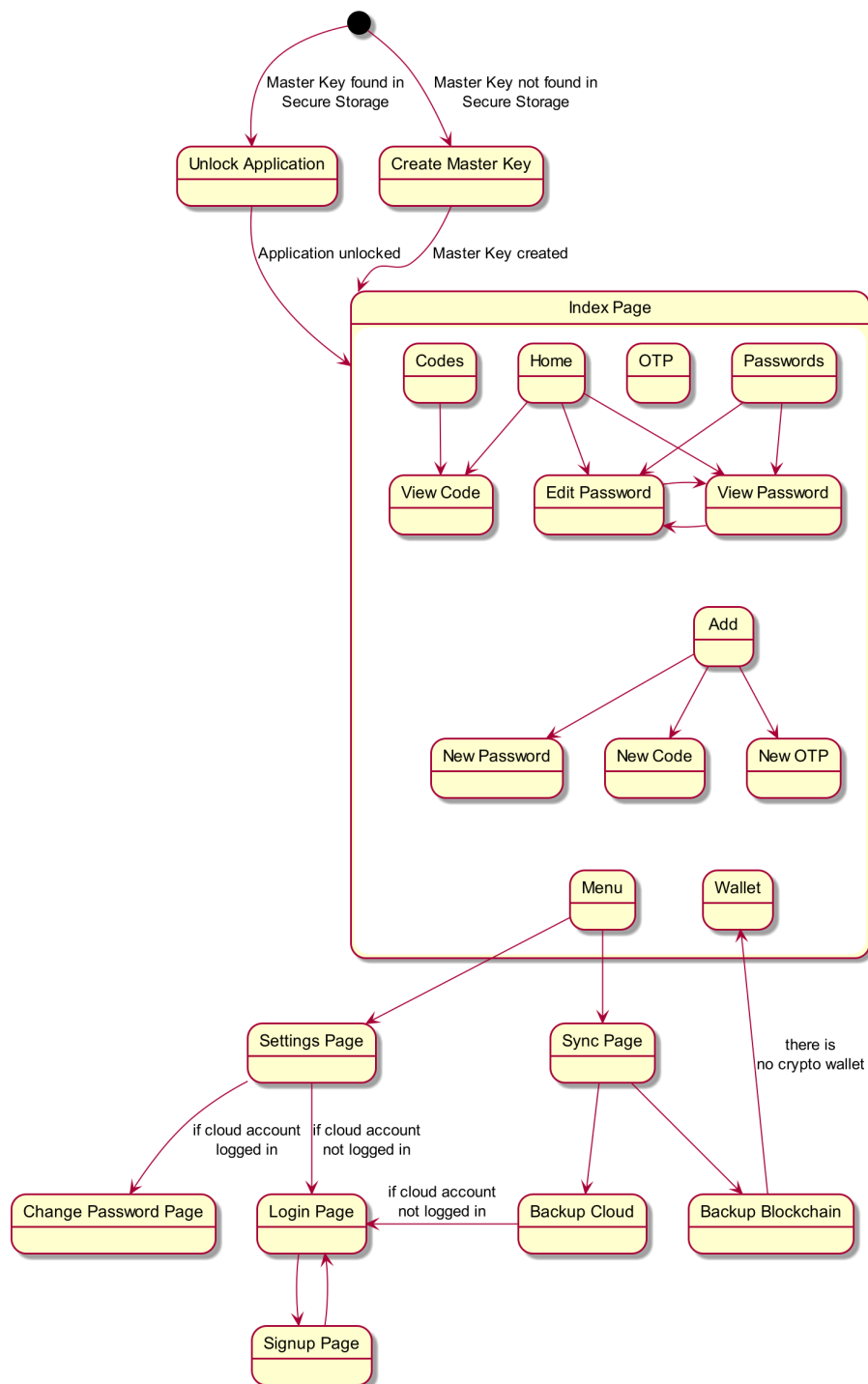


Figure 3.3: Flow diagram showing the main pages and fragments of the application

In Figure 3.3 we can see the flow between the main elements of the user interface. Initially the user needs to input the master key in order to unlock the application or create one if it does not exist.

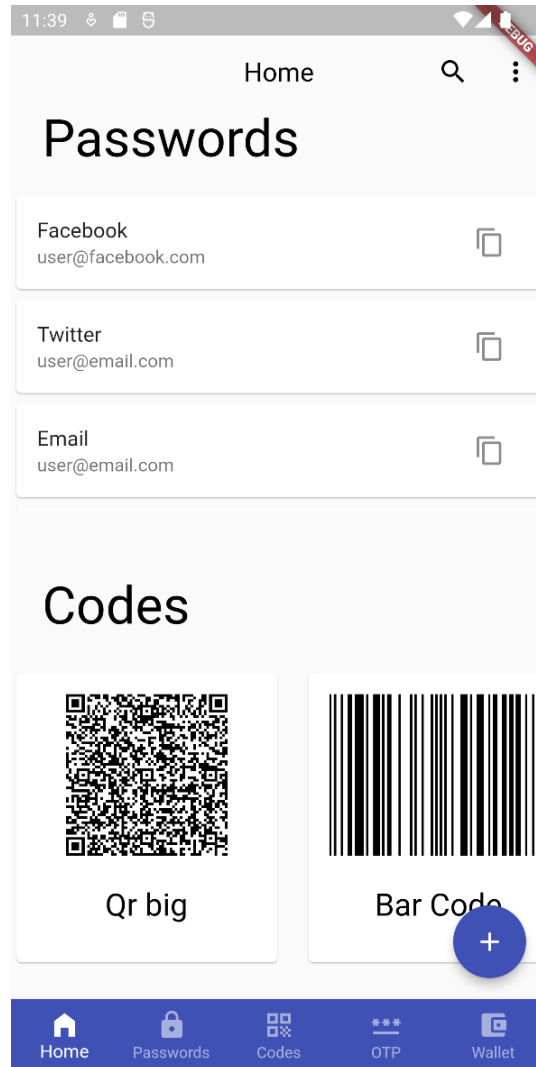


Figure 3.4: Home fragment of the application

After the application is unlocked, the user is redirected to the Home fragment, Figure 3.4, of the Index page. Here the user has quick access to passwords, codes and the wallet balance. The user can navigate different fragments using the bottom navigation bar. The passwords fragment contains a list with all the passwords. Here the user can copy the passwords if they tap on the copy button of a password. On long tap a context menu giving the user the delete, edit and copy options shows up. On tap a password opens a details page containing information about the password. From this page the user can go to the edit page where they can change the data related to the selected password.

The Codes fragment is similar with the password page. It shows a list that contains all the codes. If the user taps on a code, a details page will be opened showing a big picture of the code and its content. The user can delete the code in the details page or in the Codes fragment by log tapping and selecting the delete option in the context menu.

The OTP fragment shows a list containing all the stored tokens and the current code for

each. If the user taps on one of these list entries the code will be copied to the clipboard. The user can delete an OTP token by selecting the delete option from the context menu after a long tap on a token.

The last fragment is the Wallet. Where if there is no wallet created yet, the user can create a new wallet or import one by introducing the private key. When a wallet exists the balance of the wallet and the public address are displayed.

On the top right of the Index Page there is a menu, seen in Figure 3.4. This menu offers the user the option to open the sync page, the settings page or the password generator.

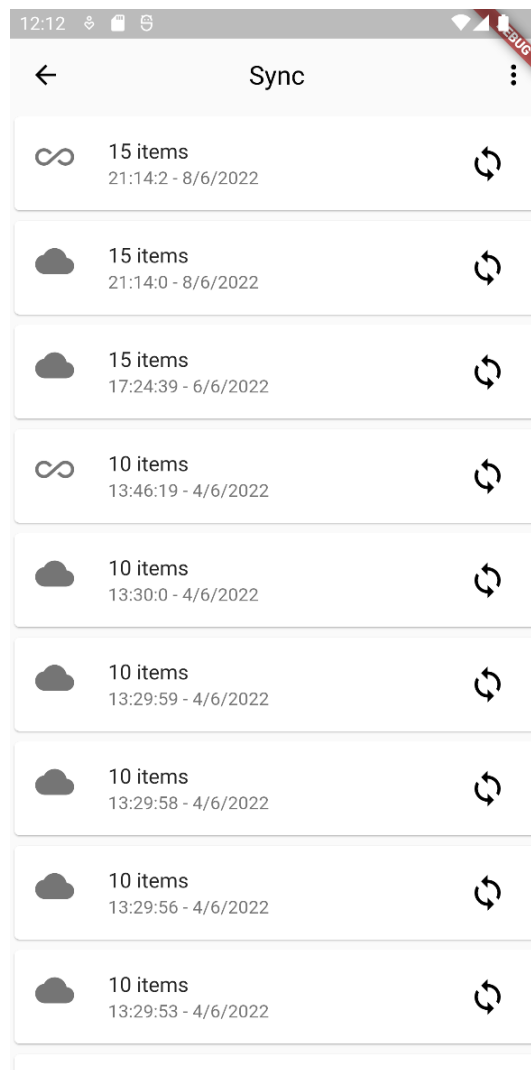


Figure 3.5: Sync page of the application

The sync page, Figure 3.5, contains a list with all the available backups the user can restore. A backup can be restored by pressing the restore button of the desired backup. Every backup has the number of items displayed and the date and time when it was created is also shown. The backups are sorted chronologically, the most recent ones being at the top. The blockchain backups are represented by an infinity icon and the cloud ones by a cloud icon.

In case there are no backups available the user is prompted with the option to create one instead of being shown an empty list. The user can also create a backup by selecting the appropriate option from the top right menu.

If the user tries to create a blockchain backup but there is no wallet set, they are redirected to the wallet fragment of the index page. Where they are prompted to create or import a wallet.

Similar, in the case when the user wants to create a cloud backup, but they are not logged into a cloud account, they are redirected to the login page and prompted to login or signup for the cloud service. After the login process is done, they are redirected back to the sync page and a cloud backup can now be created.

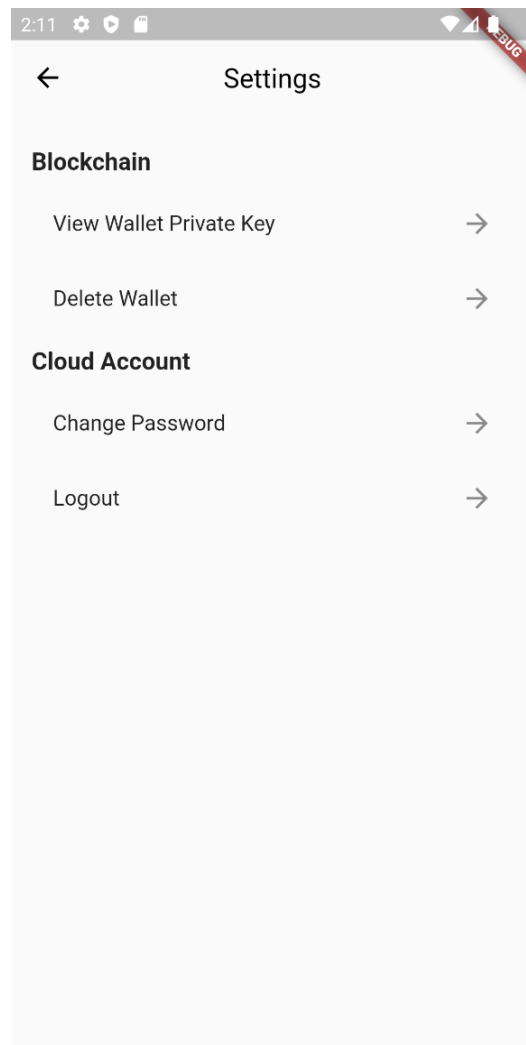


Figure 3.6: Settings page of the application

The settings page, Figure 3.6, can be accessed from the index page menu. Here the user can delete or view the wallet private key and manage their cloud account. The settings page changes depending on the state of the application. If the user did not create or link a wallet yet, the settings page will display the options to do so instead of the delete and view ones. If the user did not login to the cloud account yet the settings page will display a login option.

### 3.4 APPLICATION STATE

The application state is responsible for storing the data the application works with. This data is needed to build the user interface.

As stated in the *Technology Stack* chapter, the state management system used in this application is Redux. Unlike in other architectures like BloC, in this pattern the entire application has only one state object that contains all the information.

Each time the state is updated, the user interface is rebuilt as well. In order to selectively rebuild the user interface components, containers are used. Containers are specific providers that select only one object from the AppState class that is used to build the user interface. If there are more than a single element from the AppState needed to build a part of the user interface, nested containers can be used. Each time the container detects that the represented data has changed, it rebuilds the afferent user interface. Implementation for containers is partly provided by the flutter\_redux package[17].

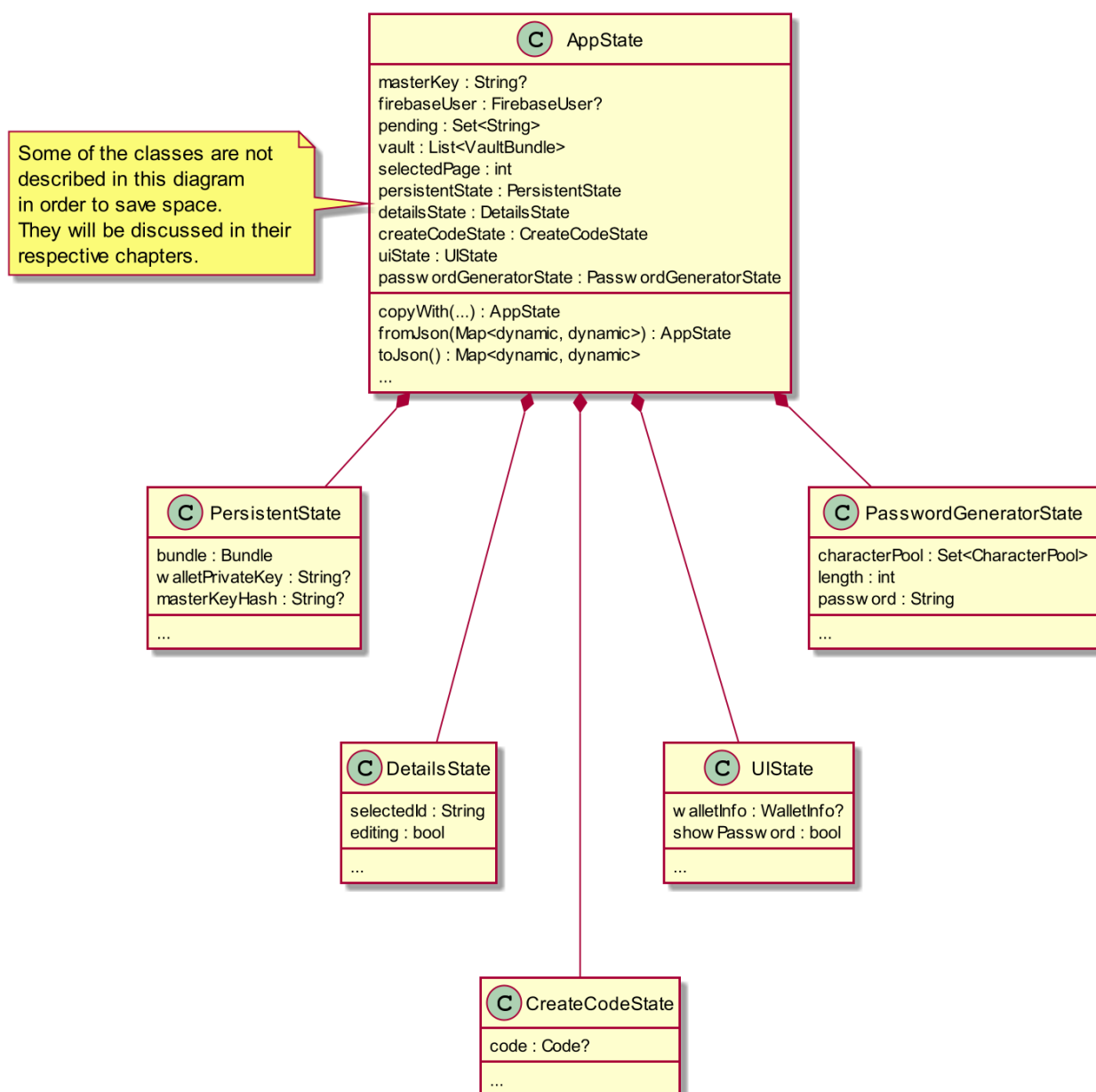


Figure 3.7: Class diagram of the AppState model

The AppState model is described in Figure 3.7. This model is composed of five other state



classes. In Dart ‘?’ next to some variables signifies that the specific type is nullable and the null value can be assigned to it.

```
@freezed
class AppState with _$AppState {
  const factory AppState({
    @Default(<String>{}) Set<String> pending,
    String? masterKey,
    FirebaseUser? firebaseUser,
    @Default(<VaultBundle>[]) List<VaultBundle> vault,
    @Default(0) int selectedPage,
    @Default(PersistentState()) PersistentState persistentState,
    // UI states
    @Default(DetailsState()) DetailsState detailsState,
    @Default(CreateCodeState()) CreateCodeState createCodeState,
    @Default(UIState()) UIState uiState,
    @Default>PasswordGeneratorState() PasswordGeneratorState passwordGeneratorState,
  }) = AppState;

  factory AppState.fromJson(Map<dynamic, dynamic> json) => _$AppStateFromJson(Map<String, dynamic>.from(json));
}
```

Figure 3.8: AppState model implementation

In Figure 3.8 is the implementation of the *AppState* model. This model is generated using the frozen package based on the template shown in Figure 3.8.

First, we have the master key in the form of a nullable string. At the start of the application, the master key is null and after it was verified with the master key hash from the persistent state, it is set to the appropriate value.

The *firebaseUser* field contains data about the cloud user account if it exists. In case the user is not logged into the cloud account the field is null.

The set of pending strings is used to signal to the user interface when some special actions processes are ongoing. These actions have a specific pendingId which is added to the set when they are dispatched and removed when the final action is completed. This helps makes it so the user interface can show progress indicators to the user when an operation is ongoing.

All the fetched backups from the cloud and the blockchain are stored inside the state into the *vault* list. The VaultBundle type contains a timestamp at which the backup was made and a ‘Bundle’ which is an object containing the passwords, OTP tokens and codes of the application at one point in time and it will be described in more detail in another chapter.

The *selectedPage* integer represents which fragment will be shown on the index page.

The persistent state is the data of the state that needs to be stored using the Secure Storage API into permanent memory so it will be loaded back when the application is restarted. In this state we keep the ‘Bundle’ with the current database, the master key hash and the private wallet key. Everything except the master key hash is encrypted with the master key itself and decrypted after the user unlocks the application.

The details state is a small state used when the password details page is displayed.

When the user scans a new QR, barcode or OTP token, the result of the scan operation is temporally stored into the *code* field of the CreateCodeState.

The UIState is used to store information that is only relevant to the user interface, and it’s not used when calling the APIs.

The last part is the password generator state which contains the options of the generator and the generated password.

All the states are immutable and in order to be changed, a new state is created. This is done using the *copyWith* method which as arguments has all the fields from the class as nullable types. If a field is nullable and it is not preceded by the **required** modifier, it is an optional field with the default value null. If the *copyWith* function does not receive a parameter, it defaults to the value that the object stores at the moment of the call is made. The method returns an object of the same type as the class it is contained in.

The *copyWith* function as well as some more utility functions like json encoders and more, are automatically generated using the *freezed*[19] package. There are no issues with having methods that are not used in the final code since Dart has the tree shaking feature which automatically drops unused code components and from the maintenance point of view there are no issue since this is generated code, tested by the package itself.

```
final Store<AppState> store = Store<AppState>(  
  ..reducer,  
  ..initialState: const AppState(),  
  ..middleware: <Middleware<AppState>>[  
    ..EpicMiddleware<AppState>(epic.epics),  
    ..], // <Middleware<AppState>>[]  
)..dispatch(const GetMasterKeyHashStart()); // Store
```

Figure 3.9: Store implementation

The store contains the *AppState* and it is responsible for triggering the user interface update when the *AppState* is changed, and it also binds the reducer, epic, state and user interface together. In Figure 3.9 the store is declared and it is linked to the other components.

### 3.5 APPLICATION ACTION

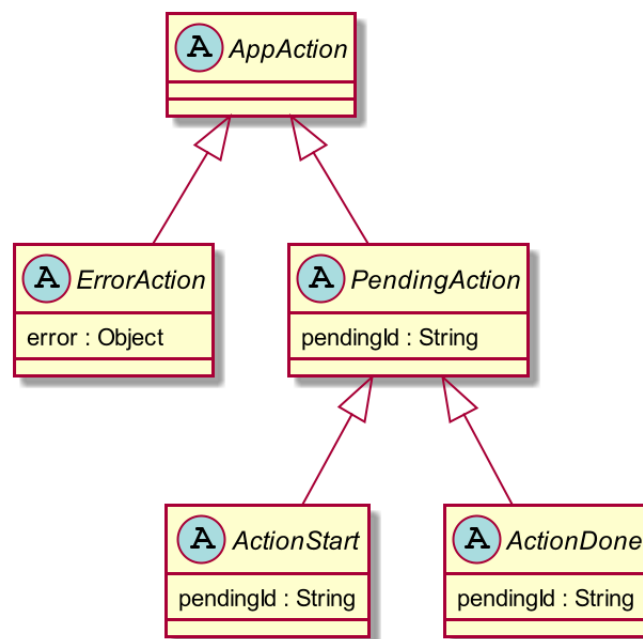


Figure 3.10: Class diagram of the AppAction model

The application action is the base type for the action events. These actions are dispatched by the user interface as result of user interaction, and they are intercepted into the epics and reducers where they trigger API calls and state change respectively.

In Figure 3.10 we can see the inheritance relationship between the action base types. The *AppAction* class is the base of all actions, and it is detected by the reducers and epics.

The *ErrorAction* class is used when an action failed due to an error. When an action stream crashes, an *ErrorAction* type object is emitted by the epic.

The *PendingAction* types *ActionStart* and *ActionDone* contain a specific *pendingId* of the action that implements them which is added or removed from the *pending* set in the *AppState* by a specific reducer.

There are three main types of actions used in this project.

The first type is the *simple action* which has only one stage. I used these for simple processes that do not require API calls like for example toggling the ‘show password’ option.

The second type are the *normal actions* which are actions that are used usually with API calls and have 3 steps: *Start*, *Successful* and *Error*. The *Start* action is the one that is dispatched by the user interface at the beginning of the process, and it contains the data needed to make the API call. The *Successful* action is the one that returns the response from the API and finishes the process. The last, *ErrorAction* is used in case the API call failed and contains an error. These actions can also have callbacks with error handling procedures.

The last type of actions used are the *pending actions* which are nearly identical to the normal ones, the only difference being that they have a *pendingId* and the *Start* action implements the *ActionStart* class and the *Successful* and *Error* actions implement the *ActionDone* classes. This type of actions is usually used with API calls that take more time to execute during which the user needs to get progress indicator feedback.

### 3.6 EPICS

The epics are the middleware part of the application which make the connection between the application state and the APIs.

An epic is basically a listener function that is executed when an action of a designated type is dispatched. It can after that make calls to the APIs and create new subsequent actions following up the first one.

Most actions in this project have epics of their own and at the application level the epics are basically a list wrapped into a specific class from the *redux-epics*[26] package.

### 3.7 REDUCERS

The reducers are very similar to the epics. They are functions that listens for specifically designated types of actions and rebuild the *AppState* based on the data contained within the actions.

One very important aspect of the redux architecture is that the reducers are the only places in which the application state should be changed. This is one of the reasons why the *AppState* class is immutable. Also, another important aspect is that reducers are executed before epics when an action is detected.

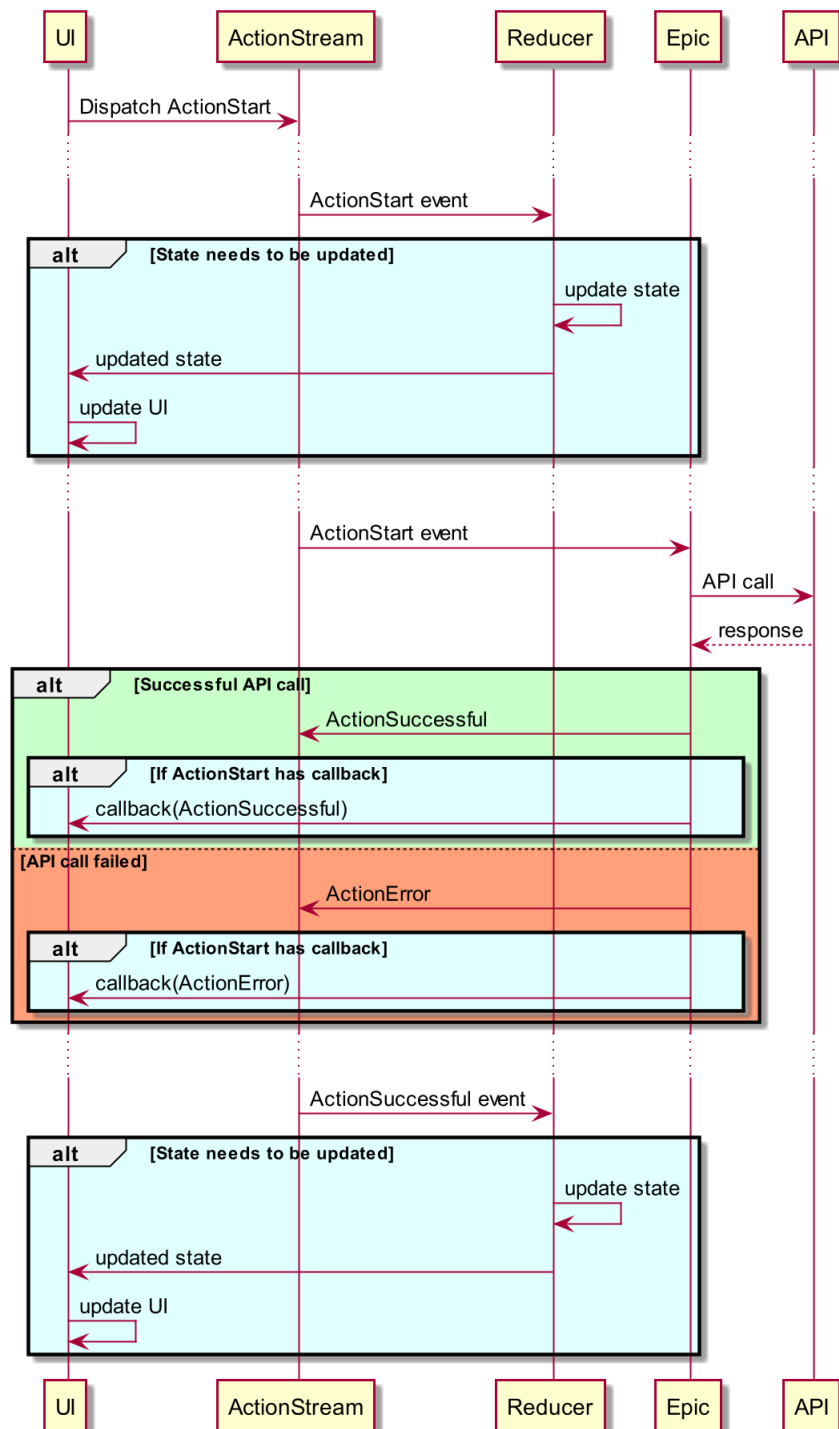


Figure 3.11: Sequence diagram of a generic normal action

In Figure 3.11 I represented using a sequence diagram how a generic normal action is processed by the application using in the redux architecture. Some of the handlers have not been included into the diagram because of size considerations but overall, this is how most of the actions are processed throughout the application.

An epic can also dispatch actions that are not part of the current process, for example when creating a new password, we want to also store it into the permanent storage, thus the stream is expanded into the afferent *Success* action and a *Start* action for the storage process.

### 3.8 APPLICATION MODELS

The data is represented by appropriate classes. All the passwords, codes and OTP tokens together create a *Bundle* of data.

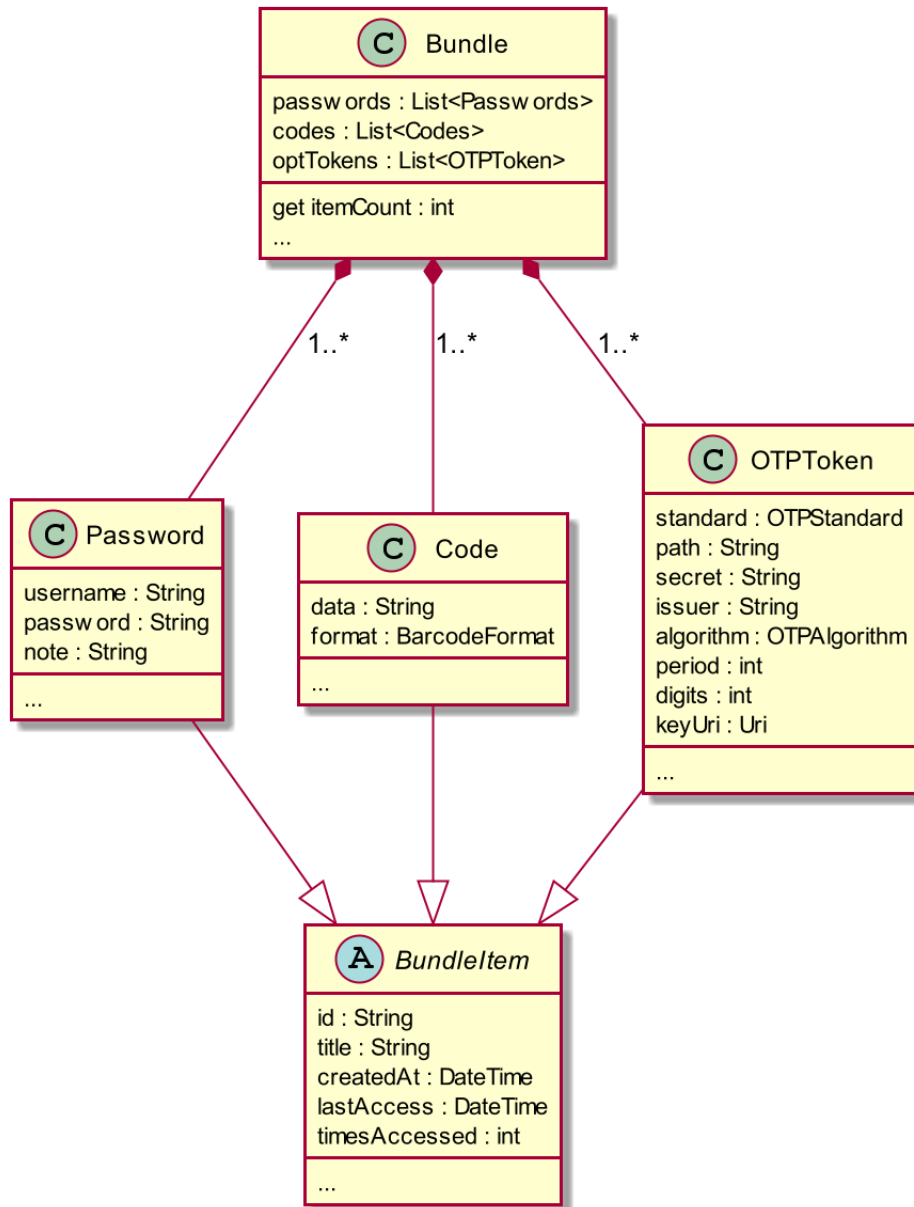


Figure 3.12: Class diagram of the Bundle model

A *Bundle* contains a list of passwords, one of codes and a list of OTP tokens as seen in Figure 3.12. The passwords, codes and tokens, all implement the *BundleItem* abstract class that contains general information about them such as an unique id based on the UUID version 4 algorithm[5] implemented into the uuid package[30], title, creation date and usage trends.

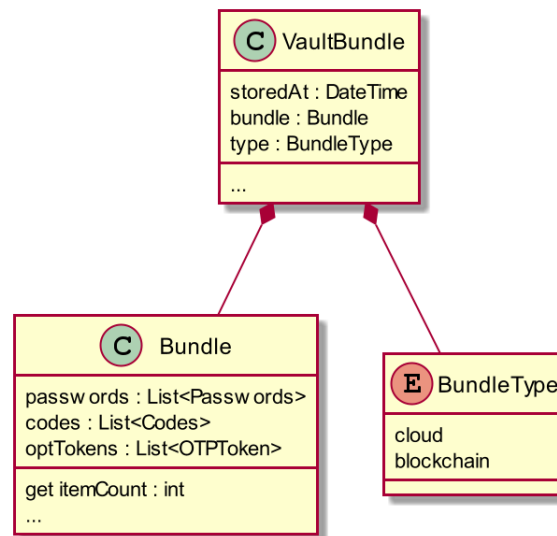


Figure 3.13: Class diagram of Bundle element in backup storage

A *VaultBundle* is what is closer to what is stored on the blockchain. As seen in Figure 3.13, in addition to the bundle, storage date and time and the type of storage are contained as well.

```

enum BundleType {
    · blockchain,
    · cloud,
}

@frozen
class Bundle with _$Bundle {
    · const factory Bundle({
    · · · @Default(<Password>[]) List<Password> passwords,
    · · · @Default(<Code>[]) List<Code> codes,
    · · · @Default(<OTPToken>[]) List<OTPToken> otpTokens,
    · · · }) := Bundle$;

    · factory Bundle.fromJson(Map<dynamic, dynamic> json) => _$BundleFromJson(Map<String, dynamic>.from(json));
}

extension BundleExtension on Bundle {
    · int getItemCount {
    · · · return codes.length + passwords.length + otpTokens.length;
    · · · }
}

@frozen
class VaultBundle with _$VaultBundle {
    · const factory VaultBundle({
    · · · required Bundle bundle,
    · · · required DateTime storedAt,
    · · · required BundleType type,
    · · · }) := VaultBundle$;

    · factory VaultBundle.fromJson(Map<dynamic, dynamic> json) => _$VaultBundleFromJson(Map<String, dynamic>.from(json));
}
  
```

Figure 3.14: Bundle model implementation

The *Bundle* and *VaultBundle* models are implemented using the *freezed* package to generate classes based on templates. The templates contain the fields the models should have, as seen in Figure 3.14, and they are used to create simple but verbose classes with utility functions such as *toJson* which encodes the model into a JSON object, *copyWith* which creates a new object based on the model it is called on and many more [19]. I also implemented a class extension to the *Bundle* model which has a function parameter that returns the sum of items in the bundle. The *BundleType* enum is used to represent where the backup bundle was stored to.

## 3.9 PASSWORD MANAGEMENT

Password management is one of the core functionalities of the application and it is the most important feature. There are multiple operations that the user can do related to password management and most of them are accessed from the passwords fragment from the index page.

### 3.9.1 Password Generator

One of the features related to the password management is the password generator.

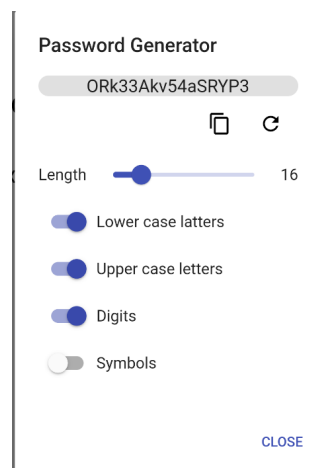
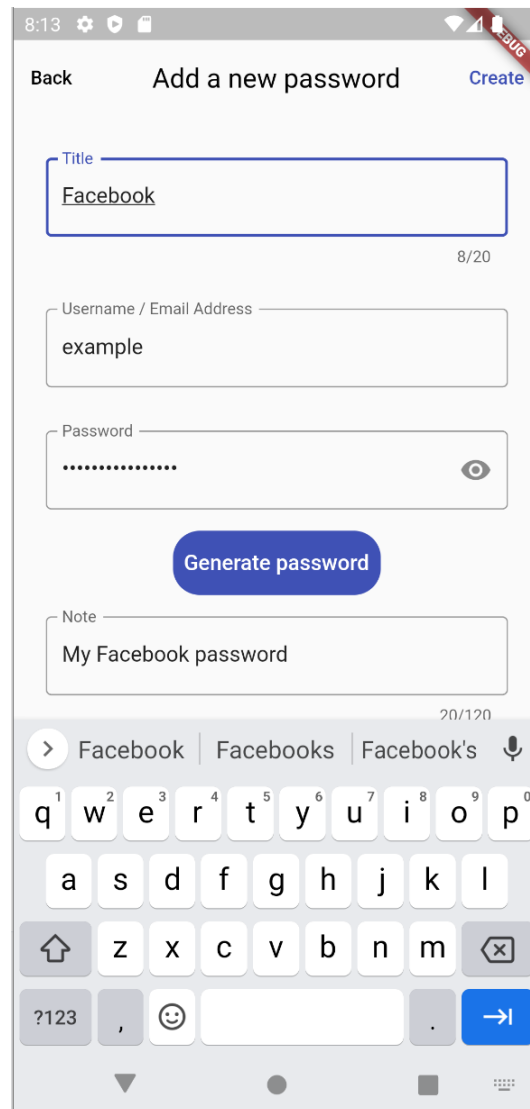


Figure 3.15: Password generator dialog

In Figure 3.15 there is a screenshot with the password generator dialog. As depicted into the image, the generator has two settings: password length and character pool. When modified these settings are saved into the state by appropriate actions. Every time the settings are changed, a *GeneratePassword* action sequence is triggered in the epics. This action can also be dispatched by the user by pressing the refresh button, and it is automatically triggered when the generator dialog is opened. Instead of an API, here the *GeneratePasswordStart* function calls a utility function with the generator settings, which returns a random password based on the parameters.

### 3.9.2 Create/Edit a Password



The screenshot shows a mobile application interface for adding a new password. At the top, there is a status bar with the time 8:13 and various icons. Below it, a navigation bar contains a 'Back' button, the title 'Add a new password', and a 'Create' button. The form consists of several input fields: a 'Title' field with the text 'Facebook' and a character count of 8/20; a 'Username / Email Address' field with the text 'example'; a 'Password' field with masked characters and a toggle icon; a 'Generate password' button; and a 'Note' field with the text 'My Facebook password' and a character count of 20/120. A keyboard is visible at the bottom of the screen, showing the text 'Facebook' in the search bar and a list of suggestions: 'Facebook', 'Facebooks', and 'Facebook's'.

Figure 3.16: New password page

When the user wants to create a new password, they have to complete a form that contains the password data. Here they have the option to use the password generator in order to create a password which will automatically complete the password field.



```
void _onSubmit(BuildContext context) {
  if (!_formKey.currentState!.validate()) {
    return;
  } else {
    StoreProvider.of<AppState>(context).dispatch(
      CreateNewPassword(
        Password(
          id: const Uuid().v4(),
          title: _title.text,
          username: _username.text,
          password: _password.text,
          note: _note.text,
          lastAccess: DateTime.now(),
          createdAt: DateTime.now(),
        ), // Password
        StoreProvider.of<AppState>(context).state.masterKey!,
      ), // CreateNewPassword
    );
    Navigator.pop(context);
  }
}
```

Figure 3.17: Function executed when the user creates a new password

When the user taps on the ‘Create’ button, the form is validated and a *CreateNewPassword* action is dispatched, as seen in Figure 3.17. This action requires a Password parameter which is also created here. After the action is dispatched, the current page is ‘popped’, the view returning to the screen where the new password creation process was triggered from.

```
@freezed
class CreateNewPassword with _$CreateNewPassword implements AppAction {
  const factory CreateNewPassword(Password password, String masterKey) = CreateNewPassword$;
}
```

Figure 3.18: The CreateNewPassword action

*CreateNewPassword* is a simple action generated based on the template from Figure 3.18. The class has two parameters, the password that will be added to the bundle and the master key that will be used to encrypt the bundle when it is stored.

```
AppState _createNewPassword(AppState state, CreateNewPassword action) {
  return state.copyWith(
    persistentState: state.persistentState.copyWith(
      bundle: state.persistentState.bundle
        .copyWith(passwords: <Password>[...state.persistentState.bundle.passwords, action.password]),
    ),
  );
}
```

Figure 3.19: The CreateNewPassword reducer

After the action is dispatched, it first is handled into the reducer where the state is updated to contain the new password as seen in Figure 3.19. Since the state is immutable, this is done using the *copyWith* function.

```
Stream<AppAction> _createNewPassword(Stream<CreateNewPassword> actions, EpicStore<AppState> store) {
    return actions
        .map<AppAction>((CreateNewPassword action) => StoreBundleStart(bundle: store.state.persistentState.bundle));
}
```

Figure 3.20: The CreateNewPassword epic

Next, the action is handled by the epic see in Figure 3.20. Here it spawns another action called *StoreBundleStart*.

```
const String _kStoreBundlePendingId = 'StoreBundle';

@freezed
class StoreBundle with _$StoreBundle implements AppAction {
    @Implements<ActionStart>()
    const factory StoreBundle.start({
        required Bundle bundle,
        @Default(_kStoreBundlePendingId) String pendingId,
    }) = StoreBundleStart;

    @Implements<ActionDone>()
    const factory StoreBundle.successful([
        @Default(_kStoreBundlePendingId) String pendingId,
    ]) = StoreBundleSuccessful;

    @Implements<ActionDone>()
    @Implements<ErrorAction>()
    const factory StoreBundle.error(
        Object error,
        StackTrace stackTrace, [
        @Default(_kStoreBundlePendingId) String pendingId,
    ]) = StoreBundleError;

    static String get pendingKey => _kStoreBundlePendingId;
}
```

Figure 3.21: The StoreBundle pending action

The *StoreBundle* action is a three step pending action as seen in Figure 3.21. In addition to the bundle that needs to be stored, the actions of this type also receive the *pendingId*. This id is monitored in the user interface and it is used to signal the process is ongoing.

The *StoreActionSuccessful* action is the ending of the process that indicates that everything went well and the API call was successful.

On the other side, the *StoreActionError* action is used when an exception was thrown during the API call.

```
Stream<AppAction> _storeBundle(Stream<StoreBundleStart> actions, EpicStore<AppState> store) {
  return actions.flatMap((StoreBundleStart action) {
    return Stream<void>.value(null)
      .asyncMap(
        (_) => secureStorageApi.storeBundle(
          action.bundle,
          store.state.masterKey!,
        ),
      )
      .mapTo<StoreBundle>(StoreBundleSuccessful(action.pendingId))
      .onErrorReturnWith(
        (Object error, StackTrace stackTrace) => StoreBundleError(error, stackTrace, action.pendingId),
      );
  });
}
```

Figure 3.22: The StoreBundle epic

This action is handled only by the epic seen in Figure 3.22. First, the epic maps the start action to the result of the *SecureStorageAPI* function call. The *SecureStorageAPI* returns a *Future<void>*, so the result can be directly mapped to the *StoreBundleSuccessful* action. In case of an exception, the *StoreBundleError* action is emitted using the *onErrorReturnWith* function provided by the rxdart package. The reason why the initial action is not mapped directly to the API call, but instead a secondary stream is created is that in case of an error only the secondary stream will crash instead of the entire application action stream.

Editing a password is very similar with the creation process. The only difference is that the first action replaces the existing password with the new one instead of just adding it to the list.

### 3.9.3 Delete a Password

The user can delete the password from many different channels. The first one is the long tap on the list element and the subsequent selection of the Delete option from the context menu. Another one is pressing the delete button in the edit page. There is also a button hidden into a slide action, similar with deleting messages into an email application.

#### Delete this item?

Are you sure you want to delete  
Facebook password?

CANCEL DELETE

Figure 3.23: Alert dialog with a warning that an item is about to be deleted

Whenever the user wants to delete a password, they are shown a warning message that the data will be lost that needs to be accepted. When the delete button is pressed a *DeletePassword* action is dispatched and similar with the create and edit ones it updates the state and triggers a bundle storage action.

### 3.10 QR AND BARCODE MANAGEMENT

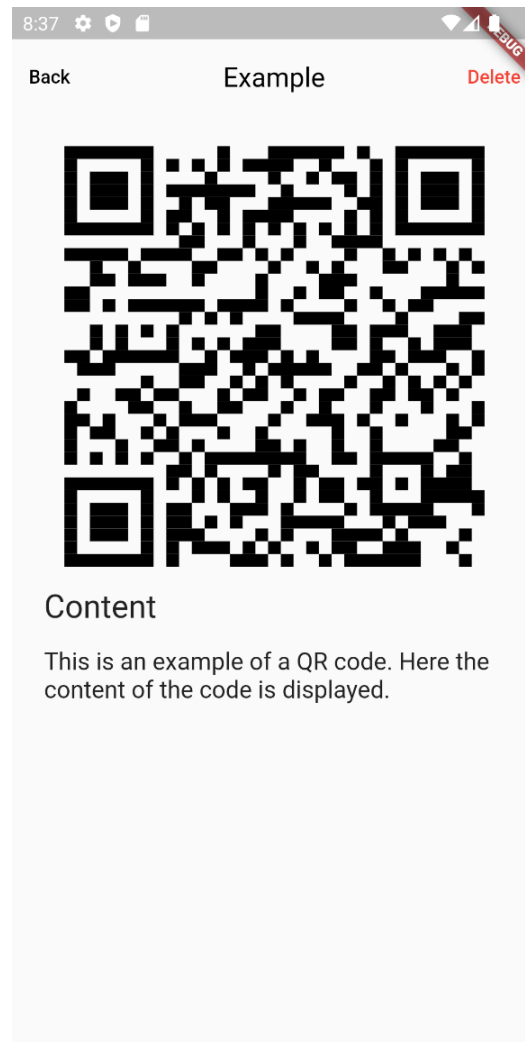


Figure 3.24: Code Details page

QR and Barcodes can be scanned, viewed and deleted. In Figure 3.24 we can see a QR code details page where the user has a big image with the respective QR code generated and the content of the code displayed.

When the user creates a new code, a dedicated page is displayed where the user is prompted to scan the code with the back camera of their mobile device. When a code is detected, the page is swapped with another one in which the code is displayed, and the user needs to add a title. After the title is added a similar process with the creation of a password happens and the code new is added to the bundle.

```
-MobileScanner(  
  ..controller: MobileScannerController(  
    ..torchEnabled: true,  
  ), // MobileScannerController  
  ..onDetect: (Barcode barcode, MobileScannerArguments? args) {  
    ..StoreProvider.of<AppState>(context).dispatch(  
      ..SetScannedCode(  
        ..Code(  
          ..data: barcode.rawValue!,  
          ..format: barcode.format,  
          ..createdAt: DateTime.now(),  
          ..lastAccess: DateTime.now(),  
          ..id: const Uuid().v4(),  
        ), // Code  
      ), // SetScannedCode  
    );  
    ..Navigator.popAndPushNamed(context, NewCodePage.route);  
  },  
), // MobileScanner
```

Figure 3.25: Scanning a code with the MobileScanner Widget

Codes are scanned using the `mobile_scanner` package [23] which offers a Flutter widget that implements the scanning functionality of a QR or a barcode for dart and it is used as seen in Figure 3.25.

```
-BarcodeWidget(  
  ..width: MediaQuery.of(context).size.width * 0.8,  
  ..height: MediaQuery.of(context).size.width * 0.8,  
  ..data: code.data,  
  ..barcode: barcodeFromScannerBarcodeFormat(code.format),  
), // BarcodeWidget
```

Figure 3.26: Scanning a code with the MobileScanner Widget

The codes are then displayed using the `barcode_widget` package [10], as see in Figure 3.25, which can generate a barcode of QR code of different types from raw data. This way only the raw data and the algorithm type are kept into permanent memory, reducing the storage size dramatically, further reducing blockchain backup creation gas cost. No images are related to QR or barcodes is stored.

### 3.11 OTP AUTHENTICATOR

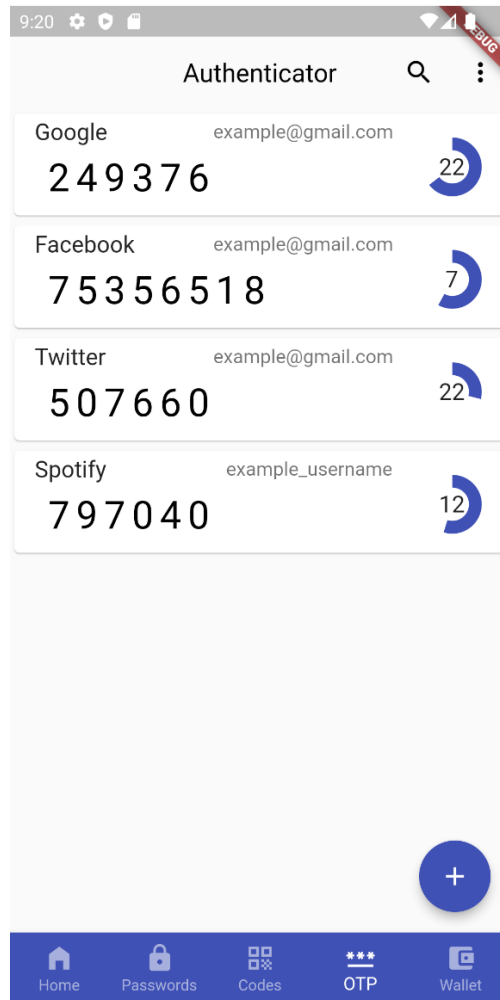


Figure 3.27: OTP fragment from the index page

The OTP authenticator functionality is the last piece of information that is stored into the data bundle and is backed up. The creation process is similar with creating a new Code but instead of entering a title, after scanning, the title is automatically deduced based on the token. In Figure 3.27 is shown the OTP fragment of the application where the list of tokens is displayed.

The OTP tokens in Figure 3.27 have different parameters and their codes are calculated correctly according to the TOTP specifications from [9].

HOTP tokens are not supported by the application. These tokens are usually used on hardware implementations that don't have an exact clock and are considered less secure. A HOTP could be converted into a TOTP, according to [9] the only difference is that the counter should be generated based on time, but there is an issue with the server implementation of such a system which is not guaranteed to increment the counter enough for it to find the current token. The implementation of a true HOTP authenticator is basically included into the implementation of the TOTP one according to [9], but I did not manage to find a standardized QR code format for this type of token.

### 3.11.1 TOTP

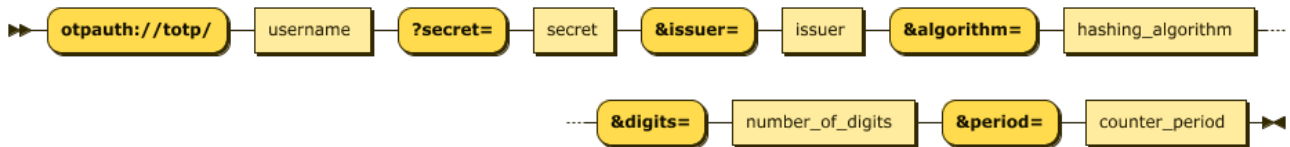


Figure 3.28: Syntax diagram showing the anatomy of a TOTP token

A TOTP token contains the information needed to create an authentication code. In Figure 3.28 we can see the structure of a TOTP token. The form of the token strongly resembles the URI format, some of the information even being represented as query parameters.

The token starts with the specification of what type of token it is. After that the username for which the token was generated is provided. This username can be an email address or just a string.

The query parameters don't have a specific order, Figure 3.28 shows just one of the possible arrangements. We start with the *secret* which is the first query parameter in this case. The *secret* is one of the inputs into the HMAC function used to generate the key. The *algorithm* parameter specifies what hashing algorithm should be used to generate the code. SHA1 is the most popular one from what I've seen, but MD5 or SHA256 are also valid algorithms. The counter is created based on the *period* query parameter using the following simple formula:

$$counter = \frac{secondsSinceEpoch}{period}$$

The *digits* query parameter specifies the length of the generated code. And at last, the *issuer* is the entity who generated the token.

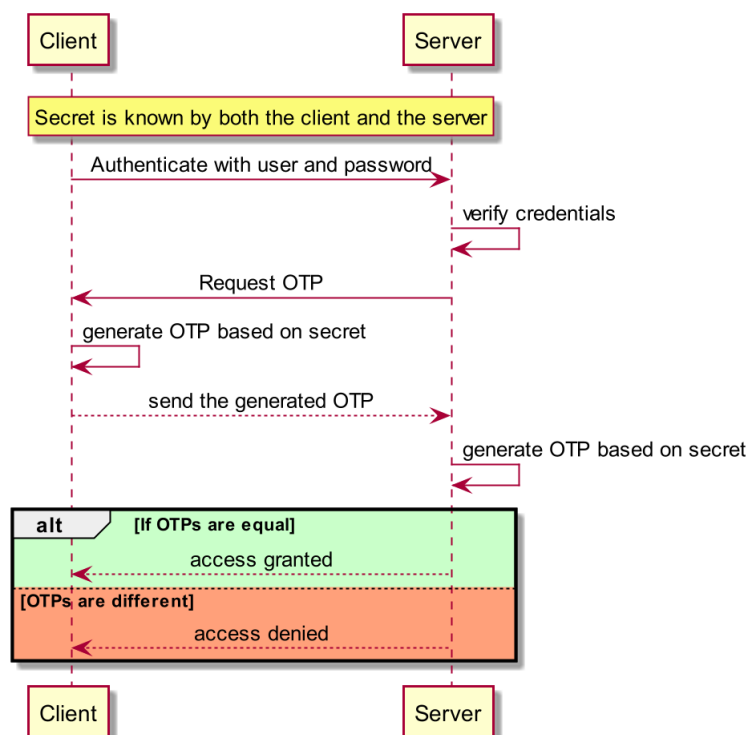


Figure 3.29: Sequence diagram showing the basic concept of OTP authentication

In Figure 3.29 I created a sequence diagram showing a basic authentication scenario based on OTP two-factor authentication. Both the client and the server have previously shared the secret with each other (by scanning a TOTP QR token) and they each generate a code which is validated by the server.

I implemented the TOTP algorithm following the specifications and technical guidance of [9].

```
String _getTOTPToken({
    ..required String secret,
    ..required int period,
    ..required int digits,
    ..required OTPAlgorithm algorithm,
    ..required DateTime time,
}) {
    ..return _getHOTPToken(
        ..secret: secret,
        ..counter: (time.millisecondsSinceEpoch / 1000 / period).floor(),
        ..digits: digits,
        ..algorithm: algorithm,
    );
}

String _getHOTPToken({
    ..required String secret,
    ..required int counter,
    ..required int digits,
    ..required OTPAlgorithm algorithm,
}) {
    ..final Hmac hmac = _getHMAC(algorithm, secret);
    ..final List<int> digest = hmac.convert(Uint8List(8)..buffer.asByteData().setInt64(0, counter)).bytes;
    ..final int offset = digest[digest.length - 1] & 0x0f;
    ..final List<int> resultBytes = digest.sublist(offset, offset + 4);
    ..final int result = (resultBytes[0] & 0x7f) << 24 | resultBytes[1] << 16 | resultBytes[2] << 8 | resultBytes[3];
    ..return (result % pow(10, digits)).toString();
}

Hmac _getHMAC(OTPAlgorithm algorithm, String key) {
    ..switch (algorithm) {
        ..case OTPAlgorithm.sha1:
            ..return Hmac(sha1, base32.decode(key));
        ..case OTPAlgorithm.sha256:
            ..return Hmac(sha256, base32.decode(key));
        ..case OTPAlgorithm.md5:
            ..return Hmac(md5, base32.decode(key));
    }
}
```

Figure 3.30: TOTP algorithm implementation

First, as mentioned before, the only difference between TOTP and HTOP is how the counter is calculated. The function *\_getTOTPToken* calls its HTOP counterpart using the appropriate counter as seen in Figure 3.30.

After that, in the *\_getHTOP* function, first the HMAC is provided by the *\_getHMAC* based on the *algorithm* parameter. The secret is converted into a byte array and then, it is fed into the HMAC function. An offset is extracted from the last byte of the digest and four bytes starting from the position equal to the integer value of the offset are taken out of the digest, they are called *resultBytes* in the implementation from Figure 3.30. These bytes are transformed into an integer and a number stored in the *digits* parameter of digits is extracted from the less significant decimal value of the resulted integer. This value is the result.



### 3.12 CRYPTOCURRENCY WALLET

The application needs to have a cryptocurrency wallet in order to create blockchain backups. Therefore, the implementation for this wallet is restricted to only spend cryptocurrency when a backup is created or deleted.

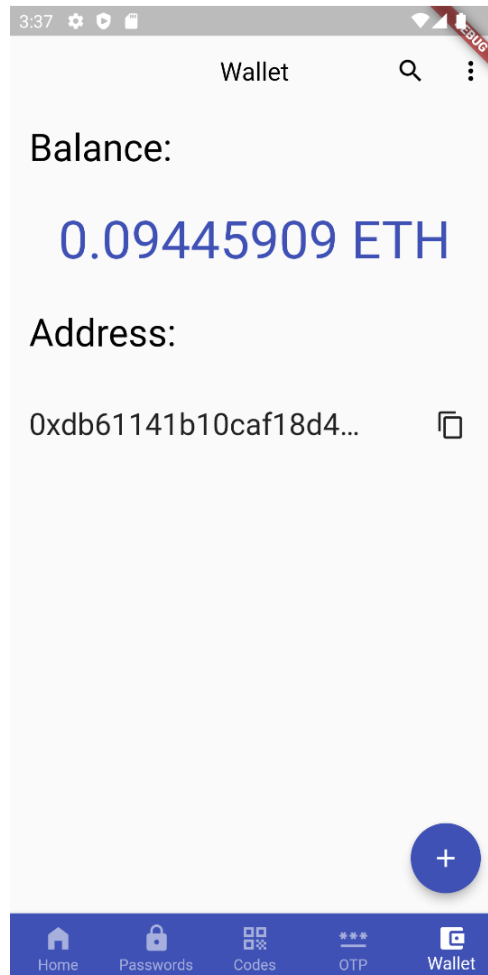


Figure 3.31: Wallet fragment from the index page

On the wallet screen, the user can see the current balance of the account and the wallet public address as seen in Figure 3.31.

```
@freezed
class WalletInfo with _$WalletInfo {
  const factory WalletInfo({
    required String balance,
    required String address,
  }) = WalletInfo$;

  factory WalletInfo.fromJson(Map<dynamic, dynamic> json) => _$WalletInfoFromJson(Map<String, dynamic>.from(json));
}
```

Figure 3.32: WalletInfo model implementation

The page can be refreshed by making a drag gesture. When an update is triggered the

*WalletInfo* object is rebuilt based on the information fetched from the blockchain.

This page refresh is also triggered automatically by a *periodic timer* every 5 seconds. This is done in order to keep the page updated in case of a balance change.

```
Future<WalletInfo> getWalletInfo({required String walletPrivateKey}) async {  
  final EthPrivateKey privateKey = EthPrivateKey.fromHex(walletPrivateKey);  
  final EtherAmount balance = await client.getBalance(privateKey.address);  
  return WalletInfo(  
    balance: (balance.getInWei().toDouble() * 0.000000000000000001).toStringAsFixed(8),  
    address: privateKey.address.hex,  
  );  
}
```

Figure 3.33: GetWalletInfo API call

The *WalletInfo* object has two string fields, as shown in Figure 3.32. The first one is the *address* which is generated from the private key. The second one is the *balance* which is given by an API call seen in Figure 3.33. This API uses the web3dart package to get the balance of the wallet based on it's public address. First the private key is created from the *walletPrivateKey* string parameter. After this, the *getBalance* method of the client provided in the web3dart package with the public address of the wallet as parameter is used to get the balance. The balance, received in the Wei Ethereum subdivision, is converted into ETH and the a *WalletInfo* model instance is built and returned.

### 3.13 BACKUP

Backing up the bundle of data is another important feature of the application. A backup item is based on the *VaultBundle* class depicted in Figure 3.13. On the cloud and blockchain respectively the backup is stored in the form of a JSON object with two fields: the *bundle* field containing the encrypted bundle and the *storedAt* field that contains the timestamp when the backup was created stored in *milliseconds since epoch*.

#### 3.13.1 Cloud Backup

For the cloud backup, the Firestore online database was used. This database is a simple key value NoSQL database. The access to the database items is made based on the cloud account that the user needs to log into before the creation of the backup.

#### 3.13.2 Blockchain Backup

For the blockchain backup part I created a smart contract and deployed it on the Rinkeby Ethereum testnet. In future this smart contract can easily be deployed on the mainnet, but for development cost reduction I decided to keep it on the testnet.

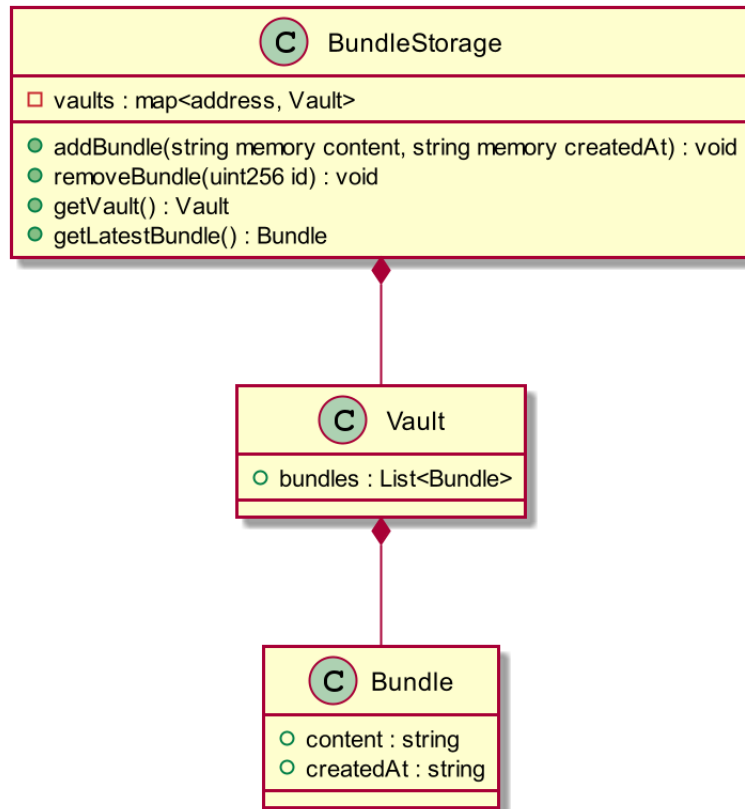


Figure 3.34: Class diagram of the smart contract

The smart contract is a simple class like data structure as seen in Figure 3.34. The contract has 4 access functions. These classes have the parameters specified in Figure 3.34 as well as the transaction details.

```

contract BundleStorage {
    mapping(address => Vault) vaults;
    struct Vault {
        Bundle[] bundles;
    }
    struct Bundle {
        string content;
        string createdAt;
    }

    function addBundle(string memory content, string memory createdAt) public {
        vaults[msg.sender].bundles.push(Bundle({content: content, createdAt: createdAt}));
    }

    function removeBundle(uint256 id) public {
        require(vaults[msg.sender].bundles.length > id, "Index out of scope");
        for (uint256 i = id; i < vaults[msg.sender].bundles.length - 1; i++) {
            vaults[msg.sender].bundles[i] = vaults[msg.sender].bundles[i + 1];
        }
        vaults[msg.sender].bundles.pop();
    }

    function getVault() public view returns (Vault memory) {
        return vaults[msg.sender];
    }

    function getLatestBundle() public view returns (Bundle memory) {
        require(vaults[msg.sender].bundles.length > 0, "No bundles available");
        return vaults[msg.sender].bundles[vaults[msg.sender].bundles.length - 1];
    }
}

```

Figure 3.35: BundleStorage smart contract implementation

From the transaction details, the most important detail for this smart contract is the public address of the sender (the mobile application wallet public address). The *vaults* are stored into a map where the key value is the public address of the sender thus everyone will have access only to their vault. The way in which the bundle is stored in the vault can be seen in the *addBundle* method shown in Figure 3.35.

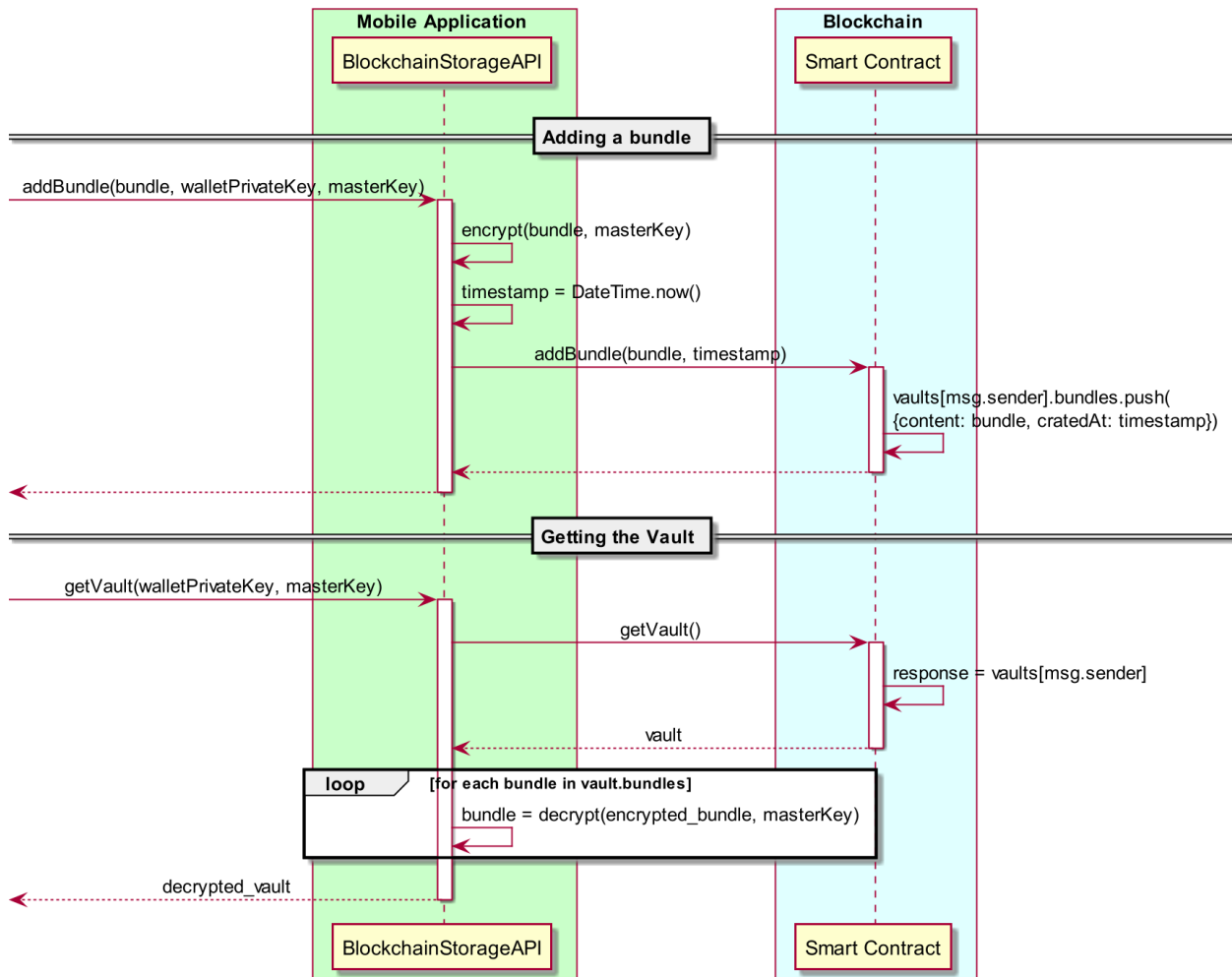


Figure 3.36: Sequence diagram showing the *addBundle* and *getVault* API calls

In Figure 3.36 I represented using a sequence diagram the two most used API calls: *addBundle* and *getVault*.

The *addBundle* API call starts when the method with the same name of the *BlockchainStorageAPI* class is called by an epic. As parameters this function receives the bundle that needs to be stored, the wallet private key used for signing the transaction and the master key that is used to encrypt the bundle. The *encrypt* static utility function encrypts the bundle with the master key and after the timestamp is fetched using the *DateTime* Dart class, the request is sent over to the smart contract. Here the contract does the simple operation of pushing the bundle to the list of bundles in the vault and then finishes the transaction as seen in Figure 3.35.

When the application requests the list of backups form the cloud, the *getVault* method of the *BlockchainStorageAPI* class is called. This method takes as parameters the wallet private key and the master key. The method then immediately sends the request to the smart contract which responds with the appropriate map entry vault as shown by Figure 3.35. When it

receives the bundle, the method decrypts each bundle from the vault using the master key. After everything is decrypted, it returns the vault to the caller.

The *getLatestBundle* and *removeBundle* contract methods are similar with the ones presented.

### 3.14 SECURITY

Since this is an application that is meant to store sensitive data, security was the top priority of the development. Every piece of information that is not in the current volatile state memory is encrypted with the master key or hashed.

```
const int _kRoundsOfEncryption = 3;

String encrypt({
  ..required String message,
  ..required String key,
}) {
  ..String response = message;
  ..final String paddedKey = _padKey(key);
  ..for (int i = 0; i < _kRoundsOfEncryption; i++) {
    ..response =
    ..    ..Encrypter(AES(Key.fromUtf8(paddedKey), mode: AESMode.cbc)).encrypt(response, iv: IV.fromLength(16)).base64;
    ..}
  ..return response;
}

String decrypt({
  ..required String message,
  ..required String key,
}) {
  ..String response = message;
  ..final String paddedKey = _padKey(key);
  ..for (int i = 0; i < _kRoundsOfEncryption; i++) {
    ..response = Encrypter(AES(Key.fromUtf8(paddedKey), mode: AESMode.cbc))
    ..    ..decrypt(Encrypted.fromBase64(response), iv: IV.fromLength(16));
    ..}
  ..return response;
}
```

Figure 3.37: Encryption and decryption utility functions

For the encryption algorithm I used AES with the master key padded into a 32 byte key as represented in Figure 3.37. Instead of implementing AES myself, I used the implementation provided by the encrypt package[13]. The data is encrypted 3 rounds. Initially I wanted to increase this number but due to data expansion, the gas prices raised exponentially, so a balanced choice was 3 rounds of encryption.

```
const int _kHashRounds = 12;
const int _kSaltSize = 6;

String hashPassword({required String password, String? salt}) {
  salt ??= generatePassword(
    characterPool: <CharacterPool>{
      CharacterPool.lowercaseLetters,
      CharacterPool.uppercaseLetters,
      CharacterPool.digits,
    },
    length: _kSaltSize,
  );
  final List<int> bytes = utf8.encode(password + salt);
  Digest digest = sha512.convert(bytes);
  for (int i = 0; i < _kHashRounds - 1; i++) {
    digest = sha512.convert(digest.bytes);
  }
  return '${base64.encode(digest.bytes)}:$salt';
}

String saltFromHash(String hash) {
  return hash.split(':')[1];
}
```

Figure 3.38: Hashing utility functions

The masterKey is stored only in the Secure Storage which itself is supposed to be encrypted by the operating system. I decided to encrypt even the data stored into the Secure Storage as an additional line of defense. The master key itself is salted and hashed using the SHA512 hashing function provided in the crypto package[11], as shown by Figure 3.38. The salt is randomly generated using the password generator functionality of the application, and the hash function is applied in 12 rounds.

Additionally, the application has strict master password requirements, the password needing to be at least 8 characters long and it must contain at least a lowercase and an uppercase letter, a symbol and a number.

## 4 TESTS

### 4.1 PERFORMANCE STATISTICS

The Flutter plugins offer statistics about the performance of the application such as RAM usage and the average FPS the application is outputting.

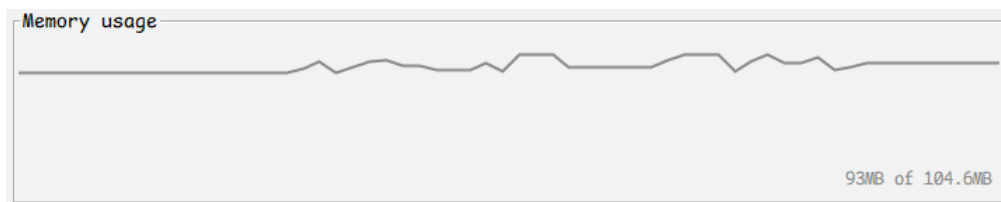


Figure 4.1: Memory graph generated by the flutter performance plugin

In Figure 4.1 there is a graph of app memory usage. On average the application consumes less than 100 MB of memory.

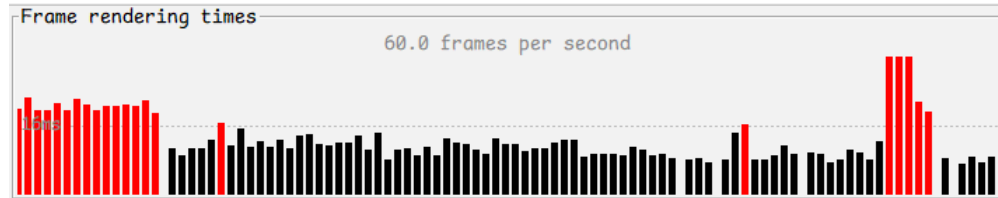


Figure 4.2: Frame rate graph generated by the flutter performance plugin

Figure 4.2 shows the average frame rate the application outputted over a period of time. As seen in the image, the application usually outputs at least 60 frames per second.

These performance metrics were taken from the debug version of the application, the code being interpreted instead of being compiled into binary code. The release version should considerably outperform these results.

## 5 CONCLUSIONS

Creating this project was a very challenging yet pleasant task. From creating a decentralized app on the blockchain to implementing a lot of different features, I believe this application has touched a lot of different subjects. Implementing a Flutter application with a Redux state management was also an interesting aspect of the project and it is something that I wanted to do for some time. In the end I plan to further develop the application, maybe add some new features and use it myself for password management.

### 5.1 LIMITATIONS

There are some limitations to the application that I did not foresee when I started implementing it. Initially I wanted to also implement an autocomplete feature which would automatically detect when a password is needed and suggest to the user to use it. Implementing this in Flutter requires an external native service since there is no access to the autofill API yet in Flutter. This task would have taken too much time to complete in the time framework I was working with, and I dropped it.

Another limitation is the gas price required to saving a backup on the blockchain. In the current Ethereum market, saving a backup on the blockchain would require a considerable amount of money (about \$100 according to my estimations).

### 5.2 POSSIBLE IMPROVEMENTS

There are a lot of possible improvements and additional features that could be implemented. The first one and most useful feature is the autocomplete functionality. This however as mentioned above would take a considerable effort.

Another improvement is the creation of a desktop application, creating a local backup on the desktop would be a useful feature. This application could also have an import function so the user could add the passwords stored in the KeePass[20] password manager.

Using another blockchain could help with the high gas prices. Monero, Litecoin or Solana are some of the options.

The crypto wallet could also be improved. The way it is implemented right now is only to pay for blockchain backup deployment. The ability to store multiple different currencies, view transactions and send money to an address would be very useful features.



## BIBLIOGRAPHY

- [1] Brooks Allen and Sarah K Bryant. The market for cryptocurrency: How will it evolve? *Global Economy Journal*, 19(03):1950019, 2019.
- [2] Andreas Biørn-Hansen, Christoph Rieger, Tor-Morten Grønli, Tim A. Majchrzak, and Gheorghita Ghinea. An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, 25(4):2997–3040, June 2020.
- [3] Ryan Bourne. The huge scale—and implications—of the private sector boycott of russia. *Policy Commons*, 2022.
- [4] Morris J Dworkin et al. Sha-3 standard: Permutation-based hash and extendable-output functions. *National Institute of Standards and Technology*, 2015.
- [5] Paul J. Leach, Rich Salz, and Michael H. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.
- [6] Rebecca M Nelson. *US sanctions on Russia: Economic implications*. Congressional Research Service Washington, DC, 2015.
- [7] Viktor Taneski, Marjan Heričko, and Boštjan Brumen. Systematic overview of password security problems. *Acta Polytechnica Hungarica*, 16(3):143–165, 2019.
- [8] Mountain View, David M’Raihi, Frank Hoornaert, David Naccache, Mihir Bellare, and Ohad Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226, December 2005.
- [9] Mountain View, Johan Rydell, Mingliang Pei, and Salah Machani. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.
- [10] barcode\_widget package documentation. Accessed: 2022-06-11, [https://pub.dev/packages/barcode\\_widget](https://pub.dev/packages/barcode_widget).
- [11] crypto package documentation. Accessed: 2022-06-11, <https://pub.dev/packages/crypto>.
- [12] Dart documentation. Accessed: 2022-06-09, <https://dart.dev/>.
- [13] encrypt package documentation. Accessed: 2022-06-11, <https://pub.dev/packages/encrypt>.
- [14] Ethereum documentation. Accessed: 2022-06-11, <https://ethereum.org/en/developers/docs/>.

- [15] Firebase documentation. Accessed: 2022-06-09,  
<https://firebase.google.com/docs>.
- [16] Flutter documentation. Accessed: 2022-06-09,  
<https://docs.flutter.dev/>.
- [17] flutter\_redux package documentation. Accessed: 2022-06-09,  
[https://pub.dev/packages/flutter\\_redux](https://pub.dev/packages/flutter_redux).
- [18] Flutter state management documentation. Accessed: 2022-06-09,  
<https://docs.flutter.dev/development/data-and-backend/state-mgmt/options>.
- [19] freezed package documentation. Accessed: 2022-06-09,  
<https://pub.dev/packages/freezed>.
- [20] KeePass product page. Accessed: 2022-06-09,  
<https://www.keepass.info/>.
- [21] KeyBase product page. Accessed: 2022-06-09,  
<https://www.keybase.io/>.
- [22] LastPass product page. Accessed: 2022-06-09,  
<https://www.lastpass.com/>.
- [23] mobile\_scanner package documentation. Accessed: 2022-06-11,  
[https://pub.dev/packages/mobile\\_scanner](https://pub.dev/packages/mobile_scanner).
- [24] PassMan product page. Accessed: 2022-06-09,  
<https://www.passman.cc/>.
- [25] redux package documentation. Accessed: 2022-06-09,  
<https://pub.dev/packages/redux>.
- [26] redux\_epics package documentation. Accessed: 2022-06-09,  
[https://pub.dev/packages/redux\\_epics](https://pub.dev/packages/redux_epics).
- [27] RememBear product page. Accessed: 2022-06-09,  
<https://www.remembear.com/>.
- [28] rxdart package documentation. Accessed: 2022-06-09,  
<https://pub.dev/packages/rxdart>.
- [29] Solidity documentation. Accessed: 2022-06-11,  
<https://docs.soliditylang.org/en/v0.8.14/>.
- [30] uuid package documentation. Accessed: 2022-06-11,  
<https://pub.dev/packages/uuid>.
- [31] web3dart package documentation. Accessed: 2022-06-11,  
<https://pub.dev/packages/web3dart>.