Soneso GmbH

Registration & Login

Wallet & ICO Portal- „Customer registration and login process”– V. 0.30

Table of Contents

1. Table of Changes 4
2. Introduction 6
2.1 Purpose 6
2.2 Intended Audience 6
2.3 Scope of application 6
3. Overall description 6
3.1 Security aspect 6
3.2 Stellar Accounts 7
3.3 Registration & Login 8
3.3.1 Registration process and creation of the user account 8
3.3.2 Login process 11
3.3.3 The user lost his password 13
3.3.4 The user wants to change his password 14
3.3.5 The user lost his 2FA secret 15
3.3.6 The user wants to reset his 2FA secret 17
3.3.7 The user lost his password and 2FA secret 18
4. Implementation details 18
4.1 General security requirements 18
4.2 Error handling 19
4.2.1 HTTP Error codes 19
4.2.2 Error response fields 19
4.3 User Messaging 20
4.3.1 Message handling inside the client 22
4.4 Multi-Language-Implementation 22
4.5 Authentication details 22
4.5.1 JWT handling 23
4.6 Registration 24
4.6.1 Registration form 24
4.6.2 Server interfaces needed to display the registration form 26
4.6.3 Preparing and encrypting the payment relevant data 27
4.6.4 Server interface for the user registration 29

4.6.5 Email-Confirmation on registration 34
4.6.6 Performing the 2FA registration 37
4.6.7 Requesting registration status 40
4.6.8 Registration flow in the client 41
4.7 Login 41
4.7.1 User provides 2FA Code at login 42
4.7.2 User does not provide 2FA Code at login 46
4.7.3 Request 2FA Secret in mobile app after login 48
4.7.4 Login flow in the client 48
4.8 Confirm token by mail 50
4.9 Confirm mnemonic 51
4.10 Password lost 52
4.11 Change password 60
4.12 2FA secret lost 63
4.13 Change 2FA Secret 68
5. Error codes and keys 72
6. Appendix 73
a. Master Key Encryption 73
1. Tables 80
2. Images 80

1. Table of Changes

Version	Description	Changed by	Changed on
0.1	First draft	Christian Rogobete	07.04.2018
0.2	Added overview diagrams	Christian Rogobete	08.04.2018
0.3	Preparing and encrypting data for registration	Christian Rogobete	09.04.2018
0.4	Definition of server interface to register new user	Christian Rogobete	09.04.2018
0.5	Diagram added regarding server side user registration, added chapter regarding email-confirmation.	Christian Rogobete	10.04.2018
0.6	Added 2FA registration	Christian Rogobete	10.04.2018
0.7	Minor improvements and work on Login process.	Christian Rogobete	11.04.2018
0.8	Improved error handling, error codes and names. Work on Login process.	Christian Rogobete	12.04.2018
0.9	Work on Login process	Christian Rogobete	13.04.2018
0.11	Improvement of interfaces and parameter names with Udo	Udo Polder, Christian Rogobete	17.04.2018
0.12	Redesign for error and language handling (still in progress)	Udo Polder	23.04.2018
0.13	Adding email confirmation logic	Christian Rogobete	26.04.2018
0.14	Improved login, completed password lost description and activity diagrams	Christian Rogobete	26.04.2018
0.15	Added implementation details for lost 2FA secret	Christian Rogobete	27.04.2018
0.15	Improved change password logic, added implementation details for change password	Christian Rogobete	27.04.2018
0.16	Added (server) functions: Login, confirm mail token, password lost, change password, 2FA lost	Udo Polder	29.04.2018
0.17	Review and update of interfaces	Christian Rogobete, Udo Polder	30.04.2018
0.18	Change 2FA secret added	Christian Rogobete	30.04.2018
0.19	Confirm mnemonic added	Christian Rogobete	02.05.2018
0.20	Alignment with Code, updated some open references	Udo Polder	03.05.2018
0.21	Request 2FA secret in mobile app after login	Christian Rogobete	03.05.2018
0.22	Added JWT-Handling	Udo Polder	04.05.2018
0.23	Added client activity diagrams for registration and login	Christian Rogobete	04.05.2018
0.24	Added email and phone number regex	Christian Rogobete	07.05.2018
0.25	Added storing of individual word list	Christian Rogobete	08.05.2018
0.26	Added some missing parameters for the encrypted user data	Udo Polder	09.05.2018
0.27	Added user message functionality	Udo Polder	14.05.2018

0.28	Cleanup document, fix client login diagram regarding error 1009	Christian Rogobete	22.05.2018
0.29	Updated registration and login client diagram. Registration to continue with login step 2 to receive full auth token.	Christian Rogobete	01.06.2018
0.30	Removed mnemonic gap, added padding to word list, minor improvements.	Christian Rogobete	17.07.2018

2. Introduction

2.1 Purpose

This specification covers the registration and login process for users who want to register and log into the portal via web client, android app or iOS app. It first describes the general processes and considerations behind it. Then it goes into detail for each component involved and describes the details to be implemented.

2.2 Intended Audience

The intended audience for this document are Soneso & partner companies project managers, server and client developers, product testers and documentation writers.

2.3 Scope of application

The app can be used as a pure wallet as well as a ICO portal with wallet.

When used as a pure wallet, the app should allow the user to register and use it for transactions. The wallet is supposed to offer the highest possible usability. For example, the user should not have to enter stellar account secrets for transactions, the wallet should persist over multiple devices, it should support the different assets provided by the stellar network and offer other features such as the possibility to create shared accounts.

When used as an ICO portal, the app should allow users to register for participating in the ICO. After registration, they can use the dashboard to complete further ICO steps such as complete the KYC process or make the payment for the ICO assets/coins. The dashboard also contains the wallet, providing the user the possibility to view the balance of her account, receive assets and submit payments to the stellar blockchain.

3. Overall description

3.1 Security aspect

In the context of crypto currencies, the security aspect plays a major role. The payment-relevant account data of the user must be particularly well protected. Hackers should not be able to use this data to make payments. The app divides the user data into two categories. The first category is non-payment relevant personal data of the user, e.g. the address or phone number of the user. The second category is payment-sensitive data that needs to be very well protected (for example, mnemonic or secret keys).

To protect the payment-relevant data of the user very well, but also to maintain the best possible usability, we have decided to use the password of the user (something that only the user knows) for encrypting his payment-relevant data. The payment-relevant data of the users is then stored encrypted in the portal database. The password of the user should never be sent to the server and thus not stored in the database. The encryption and decryption of the data takes place on the client side and thus only the user himself can access his payment-relevant data.

The advantage of this approach is that even if a hacker succeeds in hacking into the server or stealing the portal database, he cannot make any payments with the data received, because the payment-relevant data is encrypted. The data can only be decrypted with the passwords of the individual users. However, the passwords are never transmitted to the server and thus cannot be tapped. The hacker can still try to fool individual users on their own computer, but cannot get the payment-relevant data of many users at once by hacking the server or portal database.

The flip side of this concept, however, is that if the user's password is used to encrypt the payment-related data and is not transmitted to the server, then it cannot be used to authenticate and login the user into the server of the portal. Normally, in web portals, users are authenticated via their email address, password and two factor codes. However, our server cannot validate the password (because it never receives it) and thus it cannot authenticate the user by using the password.

Because we use the user's password to encrypt the payment-related data and not as usual to authenticate the users, we had to find other ways of authentication without affecting the usability of the portal, as would be the case, for example, with a second password.

In the following chapters, we will describe the solution of this problem in detail. Basically, it can be said that we internally defined a two-step authentication process. As usual, the user enters his email address, password and 2FA code to log in. In the first step, the client partially authenticates the user to the server via the entered email address and 2FA code. If these match, the server sends the encrypted data to the client, which is then decrypted by the client with the password entered by the user. The client can determine if the entered password is correct. If the password is correct, the client not only has to tell the server, but also prove it, so that the user can be completely logged in to the server in this second step and thus get access to the dashboard. How the password check in the client works and how the client can prove to the server that the user entered the correct password will be described in detail later.

3.2 Stellar Accounts

To be able to transfer ICO specific assets/coins to the user, we must create a stellar account for the user. We must configure the stellar account so that it can receive our specific assets. To create a stellar account, we need a stellar public key and corresponding seed. The public key is the users stellar account id, that is like a bank account number such as an IBAN. The seed is the associated secret key used to sign transactions. One can for example only make payments form an account if he signs the payment transaction with the accounts seed (secret key).

In the stellar network, the actual account will only be created once a payment of at least one stellar lumen has been transferred to the created public key (account id). Since this can be very costly for a large number of users, in the first step the portals client will only prepare the stellar account data during the registration process, but it will not make any payment yet to add the account to the stellar network. A payment can be made by the user himself. If it is not done by the user, this will usually be the case, it will be done by the portal as soon as the user buys specific ICO assets.

As mentioned above, the user's stellar account is prepared during the registration process. This happens in the client. In the client, the prepared, payment-relevant stellar data can be encrypted with the help of the user’s password. Only the encrypted data is transmitted to the server.

Since security and usability play a major role in our portal, we decided not only to generate a single public key and corresponding seed for the user, but to work with a 24-word mnemonic. Every registered user receives his own mnemonic from the client. It represents the equivalent of a wallet in the classical sense. The mnemonic of the user plays a central role in our portal. From the mnemonic, 231 associated public keys and seeds can be generated at any time with the appropriate stellar algorithm.

The mnemonic is automatically used by the user as backup of his wallet. It represents the wallet with all its associated accounts and secret keys. The user must write down the mnemonic which consists of 24 words during the registration process and keep them secure. The mnemonic is the most important information to be protected in the portal. It is encrypted in the client with the user’s password (which only the user knows).

The mnemonic is never sent decrypted to the server. The portal thus stores for the user the encrypted mnemonic in the portals database, but can never decrypt it on the server side, since it does not know the user’s password.

3.3 Registration & Login

To be able to use the app as a wallet or to participate in the ICO, the user must register in our portal. When registering, the portal creates a new user account. When the user logs in, he will be presented with his dashboard, which can be used for further steps, such as KYC and, upon release, the payment for the specific ICO assets. The dashboard also displays the user's stellar accounts with their balances and the user can also make payments from here. This chapter only gives you an overview of the registration and login process. Implementation details will be discussed in later chapters.

3.3.1 Registration process and creation of the user account

Following diagram shows a rough overview of the registration process:

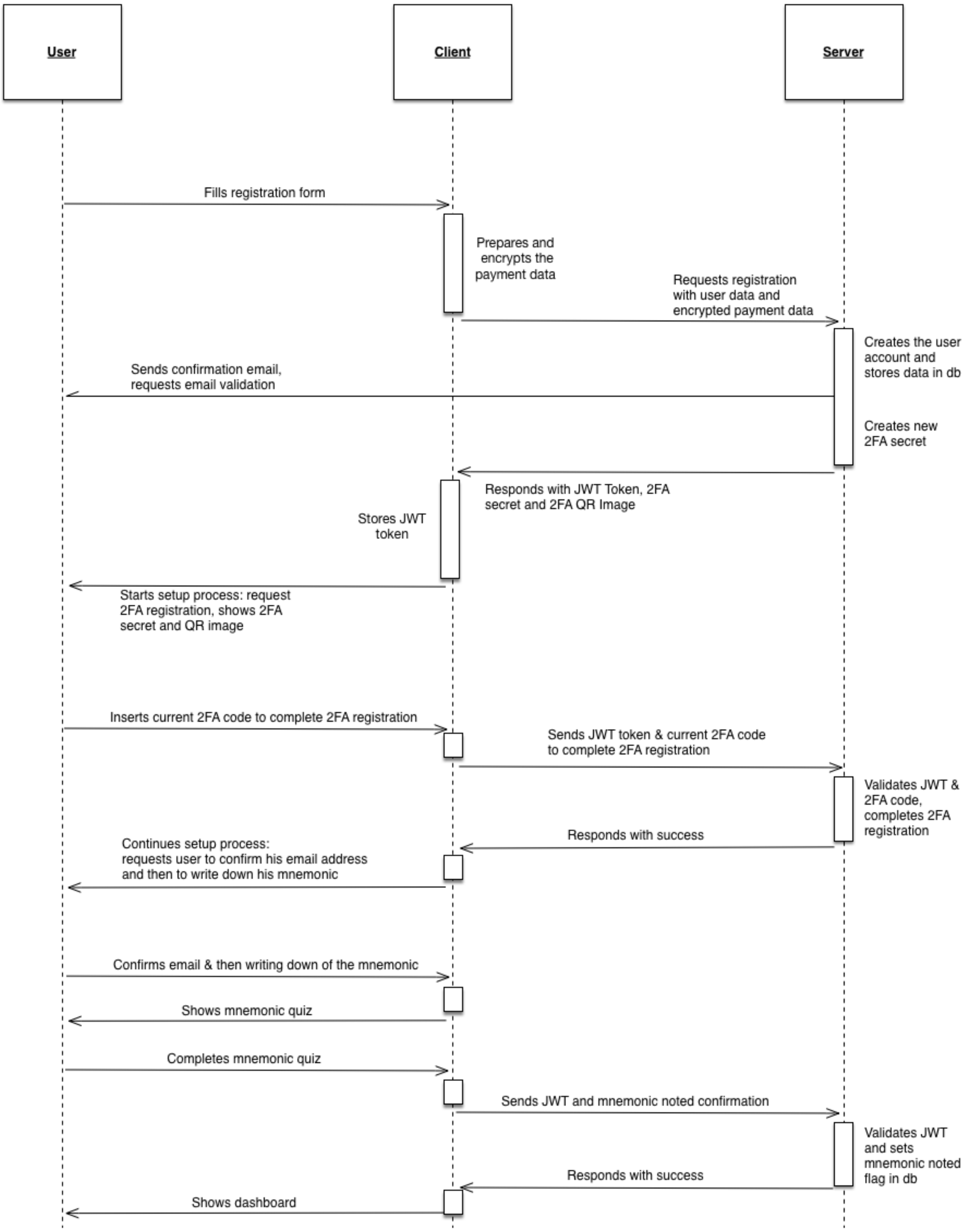

User
Client
Server
Fills registration form
Prepares and encrypts the payment data
Requests registration with user data and encrypted payment data
Creates the user account and stores data in db
Sends confirmation email, requests email validation
Creates new 2FA secret
Responds with JWT Token, 2FA secret and 2FA QR Image
Stores JWT token
Starts setup process: request 2FA registration, shows 2FA secret and QR image
Inserts current 2FA code to complete 2FA registration
Sends JWT token & current 2FA code to complete 2FA registration
Validates JWT & 2FA code, completes 2FA registration
Responds with success
Continues setup process: requests user to confirm his email address and then to write down his mnemonic
Confirms email & then writing down of the mnemonic
Shows mnemonic quiz
Completes mnemonic quiz
Sends JWT and mnemonic noted confirmation
Validates JWT and sets mnemonic noted flag in db
Responds with success
Shows dashboard

Image 1 –Rough overview of the registration process

The registration process starts in the client. Here the user enters her email address in the first step, personal data such as name, first name, address, etc., as well as the password he wants to use. The client checks the entered data for plausibility and checks the security level of the password. Only complex passwords are allowed (e.g. at least 9 characters, must contain uppercase letters, lowercase letters and numbers). If the entered data is plausible and the password is secure enough, the client will next prepare the payment-relevant data for the user.

The client first derives a 256-bit password from the user’s password. To do so, it uses a KDF with a large number of permutations. This KDF password is later used by the client to encrypt the master key. The generation of the KDF password takes, depending on the computer on which the client runs, about 1-2 seconds to complete. From a security point of view, this is important because the brute force attempt of a hacker to guess a user password is much slowed down and therefore does not make sense. The KDF password generated from the user’s password is much longer and almost impossible to guess. As described above, the client will use the generated KDF password to do more encryption, it will not use the relatively “weak” user password.

After deriving the KDF password, the client generates a random master key to be used for the encryption of the mnemonic. Please see the master key encryption concept in the appendix of this document to understand the security and usability advantages that master key encryption brings to our project.

Thereafter, the client generates a 24-words mnemonic that represents the wallet of the new user. From this mnemonic, it calculates the public key of index 0 and the public key of index 188. These are later used to authenticate the user. Next, the client encrypts the mnemonic with the master key and then encrypts the master key with the KDF password.

Thus, the client now has all the data needed to register the user in the server. It sends the user data (except password) together with the encrypted master key, the encrypted mnemonic, the calculated index 0 and index 188 public keys to the server via a secure SSL connection.

The server checks the received data and if there is not yet an account with the submitted email address, it creates a new account and stores the received data into the portal’s database. It stores the public key of index 0 in the database in plain text. The public key of index 188 is treated similarly to a password. The server stores it encrypted (bcrypted). The master key and mnemonic have already been encrypted by the client. So, the server can save them as it received them from the client.

After the server has successfully created the account, it sends a confirmation mail to the user's email address and responds to the client with the request to perform the users 2FA registration. The server hands over to the client a JWT token, the 2FA secret and its associated QR image.

The client then requests 2FA registration from the user. At the same time the user also receives the email confirmation mail from the server with the request to confirm his email address. If the user decides to perform the 2FA registration first, he must enter the 2FA code in the client. The client then sends the 2FA code and JWT token to the server. If the entered 2FA code is correct, the user is fully logged in, the client displays the dashboard.

However, there is also the possibility that the user first confirms his email by pressing the link in the confirmation mail. This link opens in a separate, second client. It thanks the user for the confirmation of the email address and redirects the user to the login form. Later, in the login process it is determined that the 2FA registration is still pending and the user is asked to complete it to be able to get to the dashboard.

If the user has completed the 2fa registration but has not yet confirmed his email, the client will ask him to confirm his email address before performing further steps. The client also shows a "Continue" button, so that the user can press this after he has confirmed his email address.

As soon as the user completed the 2FA registration and confirmed his email address, the next step is to note and confirm writing down of his mnemonic. In this setup step, the user's mnemonic will be displayed. She is asked to note it down and keep it safe. As soon as the user confirms that she wrote down his mnemonic, the client asks the user in a "mnemonic quiz" for the indexes of 4 words of the mnemonic. If the user enters the indexes correctly to prove that he wrote the mnemonic down, the client informs the server about it so that the server can set the corresponding flag in the database. However, if the user aborts the setup process, for example by closing the client, the process will be repeated at the next login, until it can be safely assumed that the user has written down his mnemonic. After completing the setup process, the user can be redirected to the dashboard. Meaning that the registration process can only be finished when the user has completed the 2FA registration, has confirmed his e-mail address and has proved that he wrote down his mnemonic.

3.3.2 Login process

The login process consists of two logical steps. However, the user does not notice, for her it looks like a normal login process in which he enters her email address, password and 2FA code.

Following diagram shows an overview of the login process:

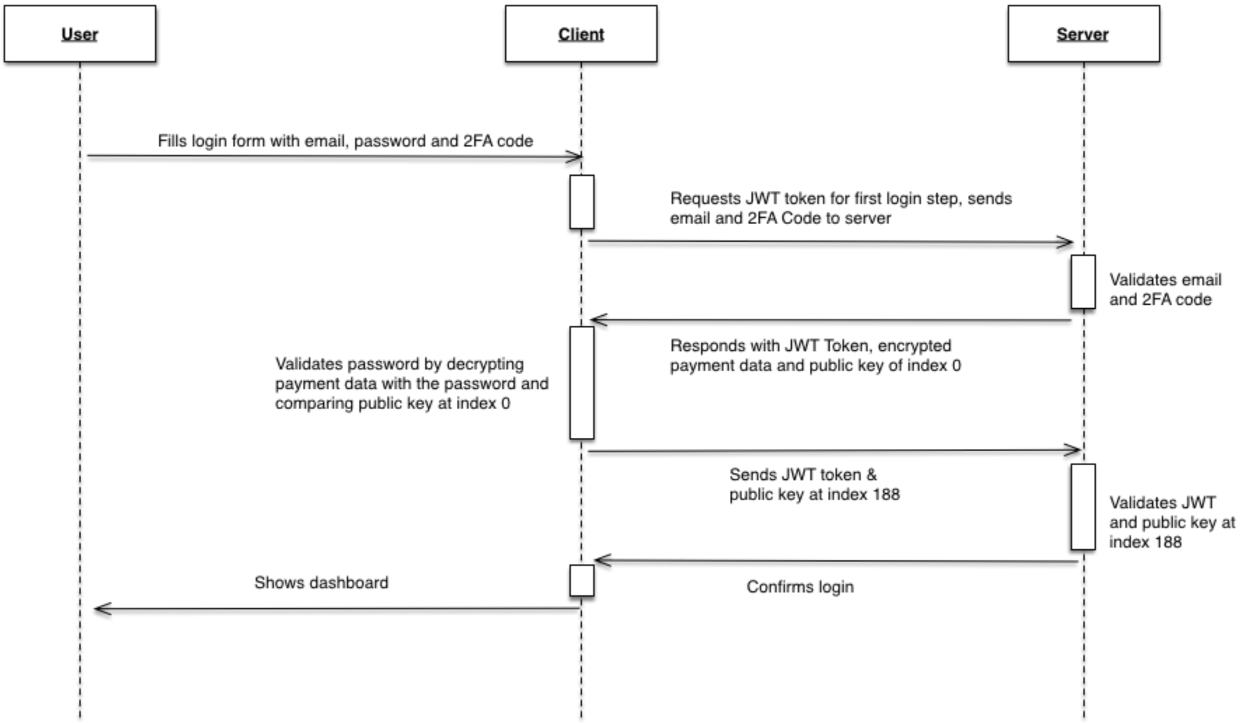

User
Client
Server
Fills login form with email, password and 2FA code
Requests JWT token for first login step, sends email and 2FA Code to server
Validates email and 2FA code
Responds with JWT Token, encrypted payment data and public key of index 0
Validates password by decrypting payment data with the password and comparing public key at index 0
Sends JWT token & public key at index 188
Validates JWT and public key at index 188
Confirms login
Shows dashboard

Image 2 –Overview of the login process

First, the user is asked by the client to enter her email address, her password and her current 2FA code. It may be that the user cannot enter a 2FA code yet. This case occurs if the user has not yet completed the 2FA registration process. Therefore, the 2FA code input field is an optional field and does not have to be filled by the user if he does not have a 2FA secret/code.

In the first step, the client sends the entered email address and 2FA code to the server to partially log in the user. The case that no 2FA code could be sent to the server because the user has not yet completed his 2FA registration process will be discussed later.

If the user could enter his email address, password and 2FA code and the server successfully validated that the email address matches the entered 2FA code, then the first logical step of the login process is completed and the server responds to the client with a JWT token (partially logged in), the user's encrypted master key, encrypted mnemonic and public key of index 0. The client can now go to second step of the login process.

In the second step of the login process, the client checks whether the password entered by the user is correct. It derives the 256 bit KDF password from the password entered by the user and decrypts the master key with the derived KDF password. With the decrypted master key, the client decrypts the mnemonic. The client then derives the public key of index 0 from the decrypted mnemonic. If the public key of index 0 determined by the client matches the one transmitted by the server, the client knows that the password entered by the user is correct.

Now the client must also prove to the server that the password entered by the user was correct. To do this, it derives the public key of index 188 from the decrypted mnemonic and sends it together with the JWT token from step one to the server. The server checks the data and if JWT token and public key of index 188 are correct, it loges in the user completely and tells the client that the user is now fully logged in by a new JWT token. The client can now forward the user to the dashboard. After showing the dashboard, the client deletes the KDF password, the public key of index 188, the decrypted and encrypted master key and the decrypted and encrypted mnemonic from its memory and does not store any of them in his internal storage/database.

As mentioned above, it may be that the user cannot enter a 2FA code at login because she has not yet completed the 2FA registration process. In this case, the client only transmits the user's email address to the server. The server then validates first whether the user has NOT completed the 2FA registration process yet. If this is indeed the case, the server sends the client a JWT token, the user's encrypted master key, the encrypted mnemonic, and the public key of index 0. Next, the client checks the user's password. If the password is correct, the client sends the JWT token and the public key of index 188 to the server. The server checks the data and if they are correct, it replies to the client with the 2FA secret and associated QR code, so the user can do the 2FA registration. If the user completes the 2FA registration and enters the 2FA code, the client sends JWT token and 2FA code to the server. The server checks the data and if the JWT token and the 2FA code are correct, then it can assume that it is the correct user (because also public key of index 188 has been validated before) and logs in the user completely. The server responds with success and the client can forward the user to the dashboard. If the user has not yet confirmed his e-mail address, then the client asks the user to confirm his e-mail before taking any further steps (setup process). If the email address of the user has already been confirmed, the client next checks if the user's mnemonic is already confirmed. If not confirmed, the client displays the mnemonic of the user and asks the user to write it down and confirm it by filling the mnemonic quiz.

3.3.3 The user lost his password

Following diagram shows an overview of the password lost process:

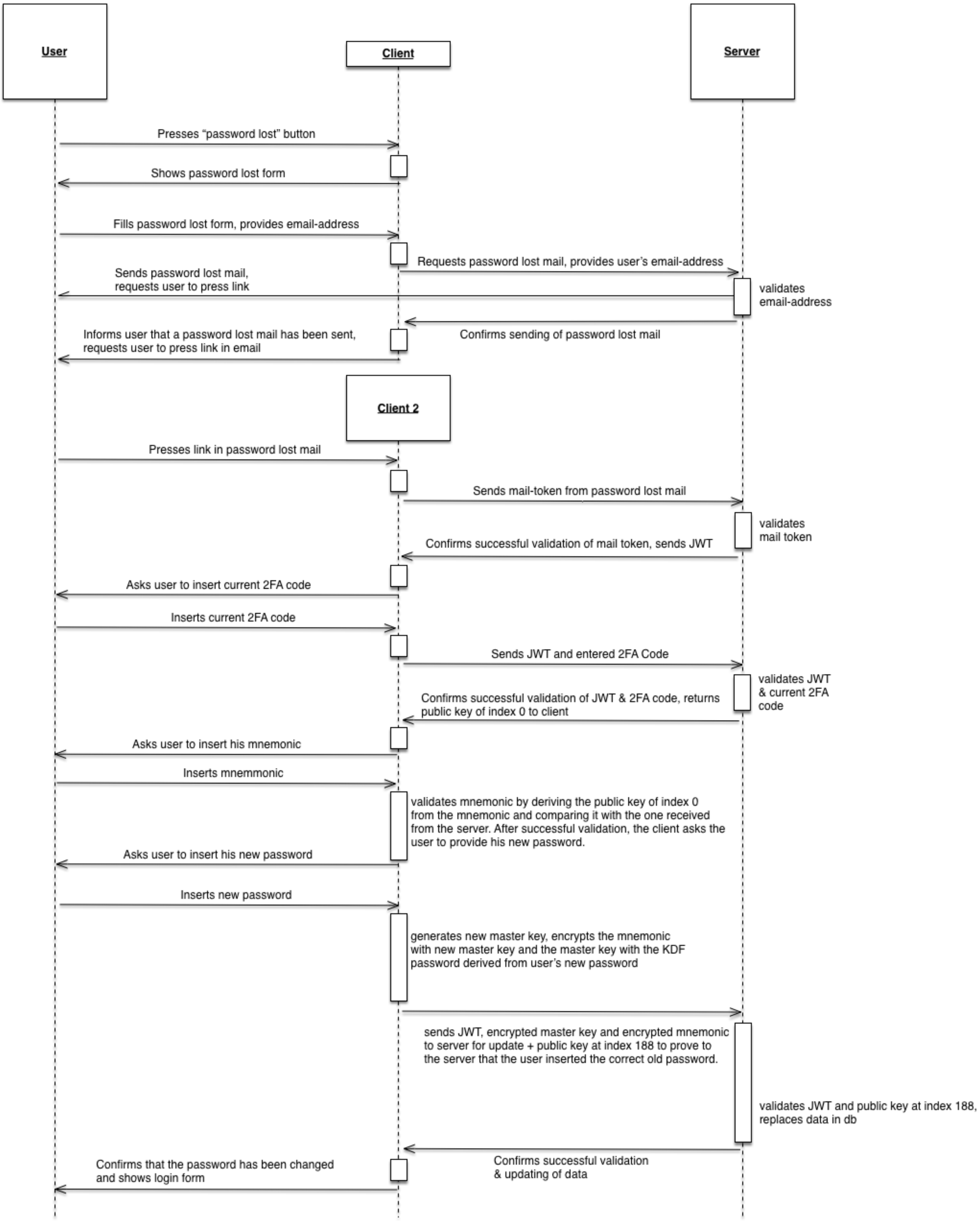

User
Client
Server
Presses "password lost" button
Shows password lost form
Fills password lost form, provides email-address
Requests password lost mail, provides user's email-address
Sends password lost mail, requests user to press link
validates email-address
Confirms sending of password lost mail
Informs user that a password lost mail has been sent, requests user to press link in email
Client 2
Presses link in password lost mail
Sends mail-token from password lost mail
validates mail token
Confirms successful validation of mail token, sends JWT
Asks user to insert current 2FA code
Inserts current 2FA code
Sends JWT and entered 2FA Code
validates JWT & current 2FA code
Confirms successful validation of JWT & 2FA code, returns public key of index 0 to client
Asks user to insert his mnemonic
Inserts mnemmonic
validates mnemonic by deriving the public key of index 0 from the mnemonic and comparing it with the one received from the server. After successful validation, the client asks the user to provide his new password.
Asks user to insert his new password
Inserts new password
generates new master key, encrypts the mnemonic with new master key and the master key with the KDF password derived from user's new password
sends JWT, encrypted master key and encrypted mnemonic to server for update + public key at index 188 to prove to the server that the user inserted the correct old password.
validates JWT and public key at index 188, replaces data in db
Confirms successful validation & updating of data
Confirms that the password has been changed and shows login form

Image 3 –Overview of password lost process

If the user forgot his password, he must first press the "forgot password" button in the client. The client then requests the user's email address and sends it to the server. If the email-address of the user and his mnemonic are confirmed, the server then sends a validation email to the user. Then, the user must press the link in the email and thereby gets to a new, second client. If the mail token from the email is correct, the server sends a JWT token (first partial confirmation) to the newly opened client. The client in turn requires the user to enter the current 2FA code. The entered 2FA code will be sent by the client to the server along with the received JWT token. The server checks the 2FA code and JWT token. If these are correct, the server tells the client about the success with and new JWT token (second partial confirmation) and sends the users public key of index 0 to the client. Then the client asks the user to enter his mnemonic. Then, the client derives the public key of index 0 from the mnemonic entered by the user and compares it with the public key he has received from the server. If the two public keys match, the client knows that the user has entered the correct mnemonic.

To be able to update the data in the server, the client must prove to the server that the mnemonic entered by the user is correct without transferring it. For this, it calculates the public key of index 188 and holds it for prove when sending the new data to the server. The client can now request a new password from the user. The user enters his new password and the client derives a new 256 bit KDF password from the entered password. The client also generates a new master key. With the new master key, the client encrypts the mnemonic entered by the user and then encrypts the master key with the KDF password.

Next, the client sends the JWT token (second partial confirmation), the new encrypted master key, and the encrypted mnemonic to the server. It also sends the public key of index 188 to prove to the server that the user entered the correct old password. The server checks the JWT token and the public key at index 188 and if they are correct, the server updates the encrypted master key and the encrypted mnemonic in the database. The new password is set. Once the server has updated the data, it responds with success. The client in turn informs the user that her new password has been set and redirects him to the login form. The user can now login with his email-address, new password and current 2FA code.

During the registration process, a gap may occur if the user forgets his password and closes the client before completing the setup process. If this occurs, the user can still reset her password. In this case, we can be sure that she did not yet use the account since she cannot reach the dashboard before completing the setup in the registration process. In this case, after validating the token from the lost password email and asking the user to input her 2fa code, the client generates a new mnemonic for the user, encrypts it with the new password inserted by the user and sends the new data for update to the server. In case that the user did not yet confirm her email address, she is asked to do so before sending the password lost email to her. If the user did not yet confirm her 2fa registration, she will not be asked to insert the 2fa code before asking for the new password and generating the new mnemonic. On next login, she will be asked to complete the setup in the registration process where the new mnemonic is displayed to her.

3.3.4 The user wants to change his password

The user can change his password via settings when logged in. Following diagram shows an overview of the change password process:

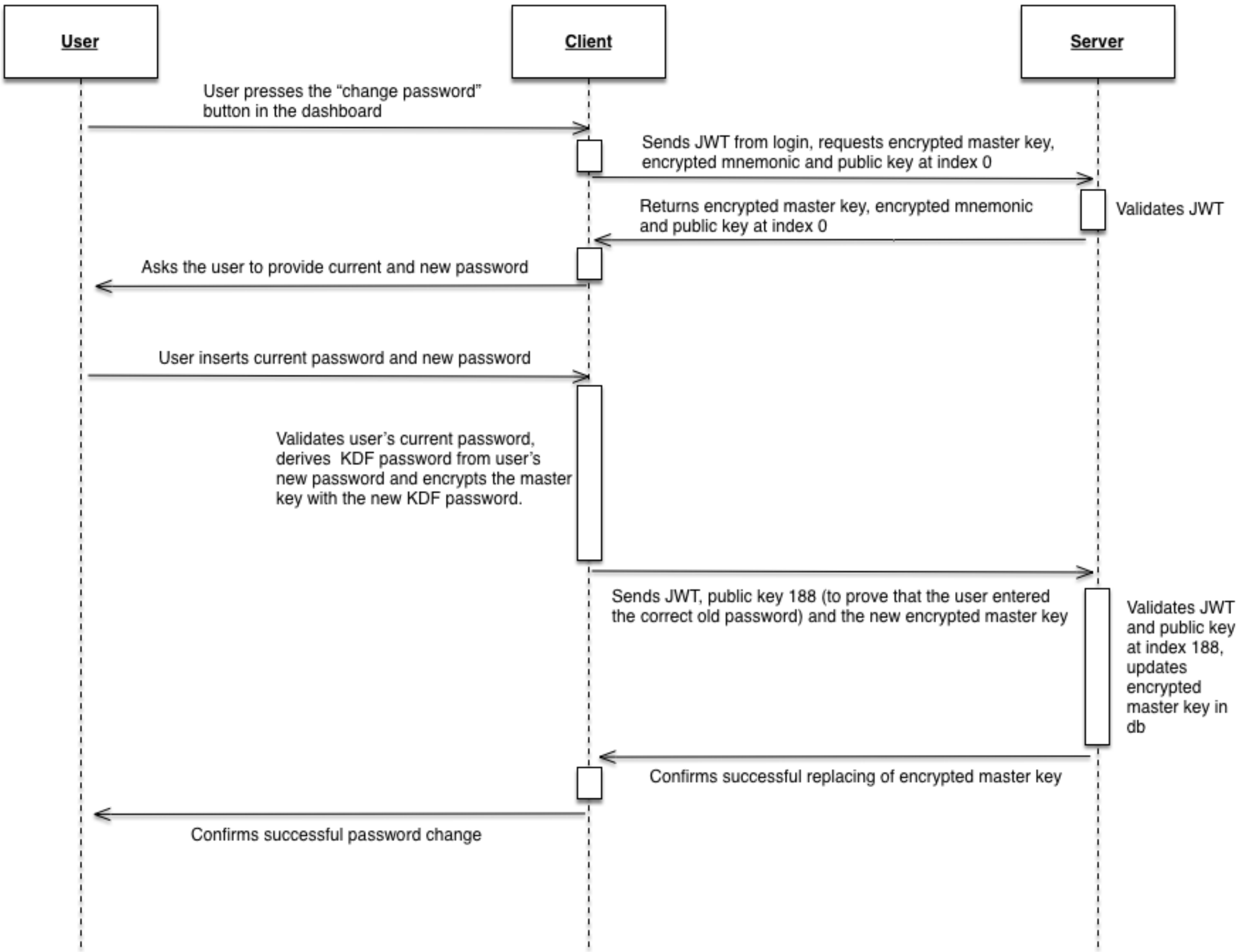

User
Client
Server
User presses the "change password" button in the dashboard
Sends JWT from login, requests encrypted master key, encrypted mnemonic and public key at index 0
Validates JWT
Returns encrypted master key, encrypted mnemonic and public key at index 0
Asks the user to provide current and new password
User inserts current password and new password
Validates user's current password, derives KDF password from user's new password and encrypts the master key with the new KDF password.
Sends JWT, public key 188 (to prove that the user entered the correct old password) and the new encrypted master key
Validates JWT and public key at index 188, updates encrypted master key in db
Confirms successful replacing of encrypted master key
Confirms successful password change

Image 4 –Overview of the change password process

To be able to change his password, the user must first log in to get to the dashboard. When the user presses the settings button to change the password, the client uses the JWT token received from the server on login to request the encrypted master key, the encrypted mnemonic and the public key of index 0 from the server. If the JWT token is correct, the server sends the requested data to the client.

Next, the client asks the user for the current and for the new password. Once the user has entered the two passwords, the client first checks the current password entered by the user. If the old password is correct, the client uses the new password to derive a new KDF password. With the new KDF password, it encrypts the master key and sends the newly encrypted master key together with the JWT token and the public key of index 188 (to prove that the inserted old password was correct) to the server. The mnemonic is not re-encrypted, it is enough to re-encrypt the master key. In addition, the client deletes the user's passwords, the KDF passwords, the public key of index 188, the decrypted and encrypted master key and the decrypted and encrypted mnemonic from its memory and does not store any of them in his internal storage/database. The server checks the JWT token and public key at index 188 and if it is correct it updates the master key in the database. Then, the server tells the success to the client and the client informs the user that his password has been changed.

3.3.5 The user lost his 2FA secret

Following diagram shows an overview of the 2FA secret lost process:

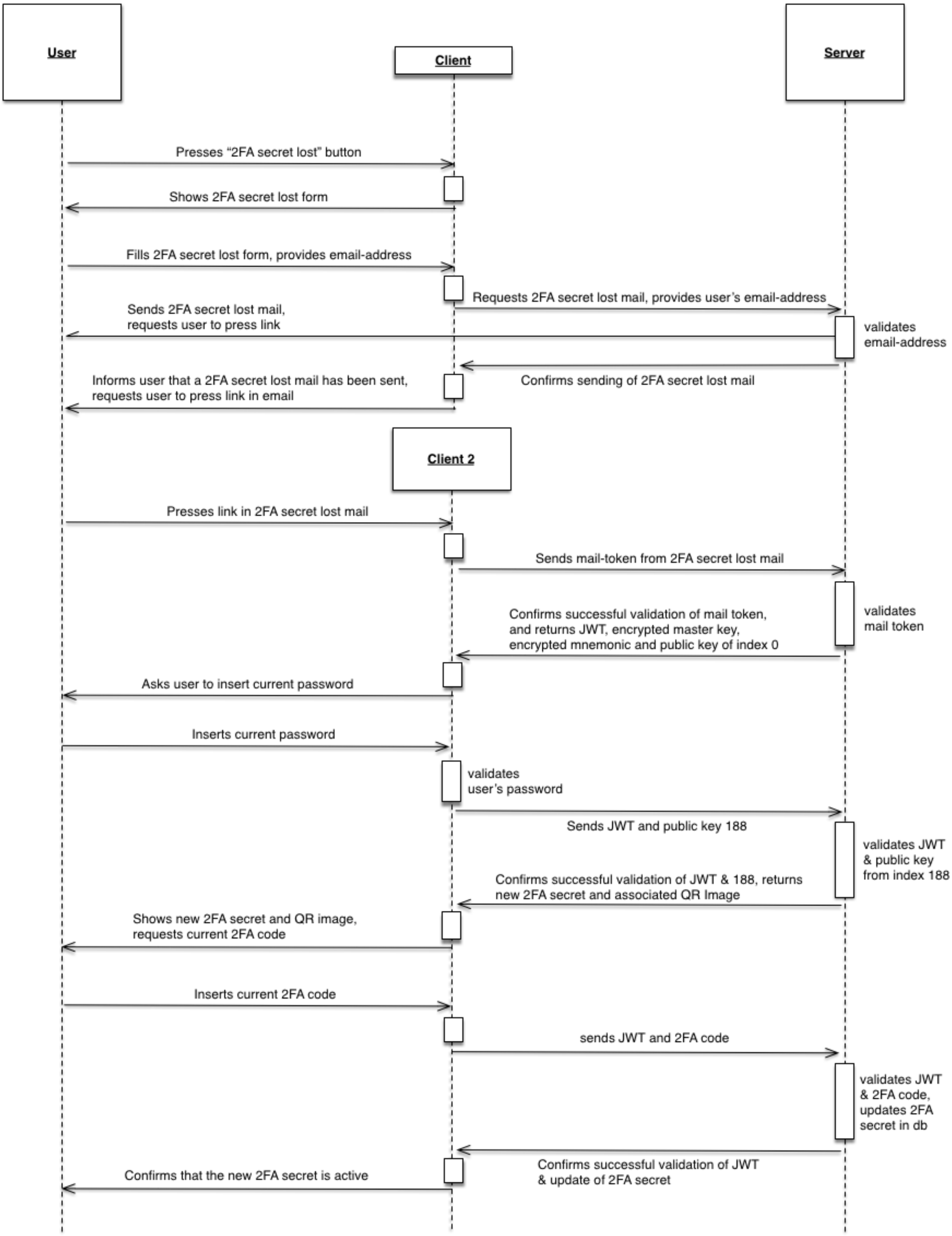

User
Client
Server
Presses “2FA secret lost” button
Shows 2FA secret lost form
Fills 2FA secret lost form, provides email-address
Requests 2FA secret lost mail, provides user’s email-address
Sends 2FA secret lost mail, requests user to press link
validates email-address
Confirms sending of 2FA secret lost mail
Informs user that a 2FA secret lost mail has been sent, requests user to press link in email
Client 2
Presses link in 2FA secret lost mail
Sends mail-token from 2FA secret lost mail
validates mail token
Confirms successful validation of mail token, and returns JWT, encrypted master key, encrypted mnemonic and public key of index 0
Asks user to insert current password
Inserts current password
validates user’s password
Sends JWT and public key 188
validates JWT & public key from index 188
Confirms successful validation of JWT & 188, returns new 2FA secret and associated QR Image
Shows new 2FA secret and QR image, requests current 2FA code
Inserts current 2FA code
sends JWT and 2FA code
validates JWT & 2FA code, updates 2FA secret in db
Confirms successful validation of JWT & update of 2FA secret
Confirms that the new 2FA secret is active

Image 5 –Overview of the 2FA secret lost process

If the user lost her 2FA secret and cannot obtain 2FA codes any more, she must first press the "lost 2FA secret" button in the client. The client then requests the user's email address and sends it to the server. If the email address of the user is confirmed, the server sends a validation email to the user. The user must press the link in the email and thereby gets to a new, second client. If the mail token from the email is correct, the server sends a JWT token (partial confirmation), the encrypted master key, the encrypted mnemonic and the public key of index 0 to the newly opened client. This in turn requires the user to enter his password.

After the user has entered his password, the client validates it and if it is correct the client must prove to the server that the user has entered the correct password. It generates the public key of index 188 and sends it together with the JWT token to the server. The server checks the JWT token and the public key of index 188 and if they are correct it sends to the client the new JWT token (full confirmation), 2FA secret and the corresponding QR image to be used for the 2FA registration.

The client shows the user the new 2FA secret and QR image and requests the current 2FA code. The user can now use the new 2FA secret and enter the current 2FA code. Then the client sends the new 2FA code along with the JWT token to the server. The server checks the JWT token and the 2FA code and if they are correct it replaces the old 2FA secret with the new one in the database and informs the client. The client informs the user that the 2FA secret has been exchanged and deletes the KDF passwords, the public key of index 188, the decrypted and encrypted master key and the decrypted and encrypted mnemonic from its memory and does not store any of them in his internal storage/database.

3.3.6 The user wants to reset his 2FA secret

The user can reset his 2FA secret via settings. To do this, he must first log in to get to the dashboard. If the user presses the settings button to reset the 2FA secret, the client asks the user for his password. Once the user has entered his password, the client uses the JWT token received from the server on login to request the encrypted master key, the encrypted mnemonic and the public key of index 0 from the server. If the JWT token is correct, the server sends the data to the client.

Now the client can check if the entered password is correct. If so, the client must prove to the server that the user has entered the correct password. It generates the public key of index 188 and sends it together with the JWT token to the server. The server checks the JWT token and the public key of index 188 and if they are correct it sends to the client the new 2FA secret and the corresponding QR image.

The client shows the user the new 2FA secret and QR image and requests the current 2FA code for the new secret. The user can now use the new 2FA secret and enter the current 2FA code. Then the client sends the new 2FA code along with the JWT token to the server. The server checks the JWT token and the 2FA code and if they are correct it replaces the old 2FA secret with the new one in the database and informs the client.

The client informs the user that the 2FA secret has been exchanged and deletes the KDF passwords, the public key of index 188, the decrypted and encrypted master key and the decrypted and encrypted mnemonic from its memory and does not store any of them in his internal storage/database.

3.3.7 The user lost his password and 2FA secret

If the user lost his password and 2FA secret, then he cannot reset both via the portal. The user must call the support hotline and identify himself. The support agent uses the admin portal to generate a new 2FA secret and gives the new 2FA secret to the user. The user can now use the new 2FA secret and continue recovering his account with the "password lost" functionality of the portal.

4. Implementation details

4.1 General security requirements

1. Client	
1.1	The web client never stores payment and security relevant data such as user's password, KDF password, master key (encrypted and decrypted), mnemonic (encrypted and decrypted), any seed or public key at index 188. If needed, the user's password must be requested from the user again, the KDF password must be derived from the user password, the encrypted master key and mnemonic must be requested form the server, the public key at index 188 must be generated from the mnemonic.
1.2	The mobile app also never stores the master key (encrypted and decrypted), mnemonic (encrypted and decrypted), any seed or public key at index 188. It can securely store the 2FA secret to improve the usability. If the user wants to use fingerprint, face recognition or similar, the app must warn the user about the decreased security level and if the user accepts the risk, the app can store the users KDF password in such a way that it can only be decrypted if the user authenticates with fingerprint, face recognition or similar.
1.3	The client and mobile apps never write anything to the logs in production mode.
1.4	When the mobile app goes to background it overlaps the currently visible view with a splash screen so that no screenshot of the current view can be taken by the system.
1.5	The mobile apps must implement ssh pinning.
1.6	To be continued.
2. Server	
2.1	The server does not read the JWT Token secret from any configuration file or similar. It always uses 2 randomly generated secrets with overlap offset of one hour. Each secret expires after 2 hours and must be automatically replaced with a new one.
2.2	The server writes only errors into the logs in production mode. It never writes user specific data such as email address or any other sensitive data to the logs. Instead it may write primary keys from the database into the logs.
	To be continued
3. Other	

Table 1 – general security requirements

4.2 Error handling

The error handling on the backend will be implemented in a generic way. Here are two examples of responses from the backend:

Good response, without error:

```
{
 "field1":"xxx",
 "field2":"yyy",
}
```

Error response (Bad-Request):

```
{
  [
   {
    "error_code" : 1000,
    "parameter_name" : "email",
    "user_error_message_key": "register.key.email",
    "error_message": "no good email"
   },

   {
    "error_code" : 1001,
    "parameter_name" : "forename",
    "user_error_message_key": "register.key.forename",
    "error_message": "no good forename"
   }

  ]
}
```

4.2.1 HTTP Error codes

Responses **without** error will always have HTTP-Status code 200. Responses **with** errors will be either HTTP-Status code 400 or 500. For invalid parameters the code will be 400. For internal server errors (like database down, gRPC call errors) the backend will return a status code of 500.

4.2.2 Error response fields

Field	Description	Example
error_code	This field represents the type of error that occurred while validating the data. All	1000

	possible error codes are listed in the chapter Error codes and keys. This field is mandatory and will be present on every error response.	
`parameter_name`	This field is optional. If this field is present and set (not empty), this means, that the error is related to a specific field. If the field is not present or empty, then the error is a generic error, which is not related to a field.	"forename"
`user_error_message_key`	This field is optional. If this field is present and set (not empty), this means, that the client should show a language specific error message, specified by the passed error message key	"register.mobile.wrong_country"
`error_message`	This field is optional. This field is used to pass in additional error describing information to the developer.	"Email already exists"

Table 2 – structure of returned error data

4.3 User Messaging

The portal implements asynchronous jobs like payment validation, KYC validation etc. which result in information for the customer. To be able to show the customer this information the system implements backend functionality, that the server can use, to signal the user, that there are new messages present for her. The backend takes care, not to do to many calls to the database, to maintain performance.

The procedure of signaling is the following:

- If a message for the logged in user is present, the server will set an http-header (X-MessageCount). This header will include the number of unread messages the user has
- If this header is set, the client must read the list of messages (/portal/user/dashboard/get_user_messages_list) and display them in an overview
- Now the user can select to show one message. The client retrieves the complete message with (/portal/user/dashboard/get_user_message) and displays it to the user. When retrieving a complete message, the server will move the message away from the current list to an archive. Therefore, the client must reread the list overview (get_user_messages_list)

Interface for reading the message list:

Method	Description	Parameters	Response
get_user_messages_list **GET** /portal/user/dashboard/get_user_messages_list	Returns a list of all user messages	**Parameters:** • from_archive ○ mandatory ○ boolean ○ specifies, if the messagelist should be taken from the archive	JSON: Array containing a list of messages { messages: [{ "id": int64, "title": string, "date_created":

			datetime }]. current_count: int64, archive_count: int64 }

Table 3 – structure of returned data for the user message list

Field description:

Field	Description	Example
`id`	ID of the message (either current or archive). Needed to read the full message	1000
`title`	Title of the message	Title 1
`date_created`	DateTime, when the message was created	2018-05-14T10:34:50+02:00
`current_count`	Number of current messages for the user	2
`archive_count`	Number of messages in archive for user	12

Table 4 – field description for the user's message list

Interface for reading one full message:

Method	Description	Parameters	Response

get_user_message
GET
/portal/user/dashboard/get_user_message
	Returns one full message	**Parameters:**
		• from_archive
		○ mandatory
		○ boolean
		○ specifies, if the message should be taken from the archive
		• message_id
		○ ID of the message
			JSON: Array containing a list of messages
			{
			"id": int64,
			"title": string,
			"message": string,
			"date_created": datetime
			}

Table 5 – structure of returned data for the user message

Field description:

Field	Description	Example
`id`	ID of the message (either current or archive). Needed to read the full message	1000
`title`	Title of the message	Title 1
`message`	Content of the message, please take care to replace \n with corresponding newline of client (e.g. in web)	Hello Udo, How are you?
`title`	Title of the message	Title 1

`date_created`	DateTime, when the message was created	2018-05-14T10:34:50+02:00

Table 6 – field description for the user message

4.3.1 Message handling inside the client

The client should intercept every call to the backend (take care, not the routes inside the client e.g. vue-router, but the API calls). If the header *X-MessageCount* is set on the returned response, the user has at least one unread message. The client now has to call *get_user_messages_list* in order to get the list of all messages and displays them. The client should store the number of unread messages in a global state and only do the call for retrieving the messages, if the count is different. Else, the call should not be done. If no header is set, the user has no unread messages, thus no call is needed.

After reading a message with *get_user_message*, the client should also update the unread messages list by calling *get_user_messages_list.*

4.4 Multi-Language-Implementation

The Portal will be multilingual and therefore the clients need to tell the server which language the current request is thought for. The server needs this information, because he must return data for that specific language and because there are functions, like sending emails, which are language specific.

To tell the server which language to use, the client will send the desired language in the header. We will use the **Accept-Language** header (see also: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html).

The client will intercept every call to the server and set the header according to the client's OS or user setting (if the user selected a specific language in the app).

On server-side, we will allow the "Access-Control-Allow-Headers" CORS-setting to be set. If the header is not set, we will use a default language (en).

When using the web client (browser) the Accept-Language header is set automatically by the browser. If the user manually changes the language in the web app, the client should set the header **Accept-User-Language** to the selected language. If this header is set, the server will use this language for handling the language. This header must also be set on **any** request, thus in the global interceptor.

4.5 Authentication details

The portal uses two different states for authentication at login. This is also reflected at the login-steps (1 and 2). The first state is used, if the user in only partially authenticated (e.g. email + 2FA code). The second state is used, when the user is fully authenticated (e.g. email + 2FA code + password correct).

The different states are distinguished by two different JWT-tokens that the server returns. The client should always hold just one token and use it for every call into the portal. The different states are also reflected in the endpoint URLs. If the URL starts with "/portal/user/xxx" there is no requirement for a token at all (anonymous calls). If the URL starts with "/portal/user/auth/xxx" the server expects a header for status partially authenticated. If the URL starts with "/portal/user/dashboard/xxx" the server expects a header for status fully authenticated.

The following list gives an overview which APIs return which type of header:

Called with state "anonymous", returns state: "partially authenticated":

Function in Server	URL
LoginStep1	/portal/user/login_step1
ConfirmToken	/portal/user/confirm_token
RegisterUser	/portal/user/register_user

Table 7 – List of endpoints returning simple authenticated header

Called with state "partially authenticated", returns state: "fully authenticated":

Function in Server	URL
LoginStep2	/portal/user/auth/login_step2
Confirm2FA	/portal/user/auth/confirm_tfa_registration

Table 8 – List of endpoints returning full authenticated header

Exception: called with state "partially authenticated" returns state: "lost password":

Function in Server	URL
LostPasswordTfa	/portal/user/auth/lost_password_tfa

Table 9 – List of endpoints returning lost password authenticated header

4.5.1 JWT handling

For security reasons, we do not want to store the secret used for generating the JWT inside our code or environment. To be able to handle this, we use two different secrets for signing the JWTs. Both keys are stored in the database and have a valid-to timeframe. Once the valid-to time for the first secret is reached, we delete the first one, move the second to the first position and generate a new secret for the second one. For both secrets, we calculate a new valid-to timeframe: one hour for the first, two hours for the second. When the client is requesting a secured endpoint, we check in the server if the signature of the JWT can be decrypted. If yes, the token was ok. If not, the client sent a wrong JWT to the server (expired, bad data, etc.). We also implemented "expiry" on the JWT.

For the client to be able to handle the "rolling" JWTs, the server offers a refresh API endpoint for every different authentication level (partially, fully, lost password). Following table shows the calls:

Interface in Server	**URL**
For partially authenticated	/portal/user/auth/refresh
For fully authenticated	/portal/user/dashboard/refresh
For lost password	/portal/user/auth2/refresh

The endpoints must be called with "GET" and the corresponding header ("Authorization") must be set. The server will then return JSON:

```
{
        "token":  tokenString,
        "expire": expire.Format(time.RFC3339),
}
```

This is the new token that must be used by the client for the subsequent calls to the corresponding secured server interfaces. The client does not evaluate "expire". The server will always return the token with the highest valid-to timeframe, which means, the token will be valid max two hours.

The client implements a logic to refresh the token in a periodical cycle.

For example, the web client implements it as follows:

- Intercepts every vue – route
- If the last update occured more than 5 minutes before intercepting the route, the client refreshes the token, otherwise it doesn't refresh the token
- The client takes care, that the different security "levels" (partially authenticated, fully authenticated, lost password) can change, e.g. while logging in. Therefore, it keeps track, which level is the current one, and refresh accordingly.

4.6 Registration

4.6.1 Registration form

To use the portal, the user must first register. For the user registration, the client can request following data from the user:

Input field	Properties	Description	Example
email	mandatory	The user's email address is required by the portal to identify the user later. Also, various emails are sent to the user. The email address must be unique throughout the portal. There cannot be two or more accounts with the same email address.	john.doe@example.com
password	mandatory	The user's password is required by the portal to encrypt its payment-sensitive data and to	01Zulu!88

		authenticate the user in the client.	
User specific data	-	Not every portal requires the user-specific data of the user. Also, some of the fields may be needed to identify the user in specific support cases. Therefore, it should be configurable in the client which of the fields are added to the form and which of them are mandatory.	Address of user
Forename	configurable	Forename of the user.	John
Last name	configurable	Last name of the user.	Doe
Company name	configurable	User’s company.	Apple Inc.
Salutation	configurable	Salutation for the user. E.g. Mr., Mrs., Dropdown, depends on uses language.	Mr.
Title	configurable	Academic title. This should only be used, if the title is part of the name.	Dr.
Street address	configurable	The user’s street.	Bronx Park
Street number	configurable	The user’s street number.	291 A
Zip code	configurable	The user’s zip code	10468
City	configurable	The user’s city	New York
State	configurable	The user’s state.	Bavaria
Country	configurable	The user’s residence country.	USA
Nationality	configurable	The user’s nationality. Uppercase two-letter code as defined in ISO 3166	DE
Mobile Phone	configurable	The user's mobile phone number.	+49 151 42 33 21 25
Birth day	configurable	The user’s birthday in ISO 8601 format: YYYY-MM-DD	1985-12-28
Birth place	configurable	The user’s birth place	Ulaanbaatar

Table 10 – possible registration data

The input of the user needs to be validated. Following tables describes the validation rules:

<table>
<tr><th>Input field</th><th>Validation rules</th></tr>
<tr><td>Email</td><td>Regex:
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])</td></tr>
<tr><td>Password</td><td>- Passwords must contain minimum 9 characters
- Passwords must contain minimum one upper case letter
- Passwords must contain minimum one lower case letter
- Passwords must contain minimum one number</td></tr>
<tr><td>User specific data</td><td></td></tr>
</table>

Forename	- Forenames can only contain letters - Forenames must be 64 characters or less.
Last name	- Last names can only contain letters - Last names must be 64 characters or less.
Company name	- Company names must be 64 characters or less
Salutation	- No client side validation required, dropdown - Server side must validate that the salutation exists in the db
Title	- Titles must be 64 characters or less.
Street address	- Titles must be 128 characters or less.
Street number	- Street numbers must be 32 characters or less.
Zip code	- The postal code must be in the format 99999 or 99999-99999
City	- Cities must be 64 characters or less.
State	- States must be 64 characters or less.
Country	- Countries must be 64 characters or less.
Nationality	- No validation required on client side, dropdown - Server side must validate that the nationality code exists in the db (Uppercase two-letter code as defined in ISO 3166)
Mobile Phone	- Apply mobile phone validation regex: ^[+]?[0-9]{11,16}$ Before applying the regex the client remove white spaces, ")", "(" and "–" from the inserted phone number. It sends the "stripped" and validated version to the server.
Birth day	- Must be ISO 8601 format: YYYY-MM-DD
Birth place	- Like city

Table 11 – validation of registration data

4.6.2 Server interfaces needed to display the registration form

To be able to fill specific dropdowns, the client requested the corresponding data from the server. The following table shows an overview of the APIs the server needs to provide:

Method	Description	Parameters	Response
salutation_list **GET** /portal/user/ salutation_list	Returns a list of possible salutations depending on the client's language	1. Language in header	JSON: Array containing a list of salutations {["Mr.", "Mrs." ..]} if country form parameter doesn't exist or is not available, the server returns EN.
country_list **GET**	Returns a list of country names depending on the client's language	1. Language in header	JSON: Array containing a list of key value pairs, key is the two-letter country code as defined in

/portal/user/ country_list			ISO 3166, value is the country name for the language given by parameter. Key and Value are separated by “:” {[{“code”:”DE”, “name”:”Deutschland”}, {“code”: “US”, “name”: “United States of America”}, …]}

Table 12 – server interfaces needed to display registration form

4.6.3 Preparing and encrypting the payment relevant data

As already described in the overview, when registering a new user, the client must prepare his payment-relevant data and transmit it encrypted to the server. This chapter describes which data are needed and how they are prepared and encrypted.

1. **Generate Mnemonic and related public keys**

First, the client needs to generate the mnemonic for the user. The implementation of this generation must be based on the algorithms defined by the stellar foundation. A description and example implementations can be found here: https://github.com/stellar/stellar-protocol/blob/master/ecosystem/sep-0005.md

The web-client implements this in java script. It uses the node example implementation as a basis. The android apposes our already implemented java mnemonic library which can be found in this Soneso repository: https://armada-it.repositoryhosting.com/git/armada-it/sjm.git . The iOS app uses our public stellar swift SDK which can be found here: https://github.com/Soneso/stellar-ios-mac-sdk. The sample iOS wallet mentioned in the readme file of the SDK already contains an example implementation that can be used.

It is important that the generated mnemonic is a 24 words mnemonic. The implementation is tested against the examples and results described by the stellar foundation (https://github.com/stellar/stellar-protocol/blob/master/ecosystem/sep-0005.md). Before randomly picking the mnemonic words from the wordlist, the client will randomly shuffle the wordlist. Later it will send the encrypted list of the selected word indexes - which represent the mnemonic - and the encrypted (individual) shuffled word list to the server. Each index must have 2 bytes. The list of indexes representing the mnemonic is kept in binary format and has 48 bytes.

After generating the mnemonic for the user, the client needs to derive the accounts at index 0 and index 188, hold the public keys (but not the seeds) in memory to be able to transmit them to the server during the registration request.

2. Generate random master keys

Next two random master keys must be generated by the client. The key must be 256-bit to be used for "AES" encryption. This key will be used the encrypt the mnemonic (list of word indexes from the shuffled wordlist) and the individual shuffled wordlist.

3. Encrypt the mnemonic and shuffled word list

Next the client needs to encrypt the mnemonic (list of word indexes from the shuffled wordlist) and the shuffled word list with the corresponding, generated random master keys. To encrypt the mnemonic and wordlist, the client uses the cipher algorithm "AES/CBC/NoPadding". It first creates two new random initialization vectors (IV) having a length of 16 bytes (from now on named mnemonic encryption IV and wordlist encryption IV). Then it encrypts the mnemonic with the generated random mnemonic master key using the created mnemonic encryption IV. It also encrypts the wordlist with the wordlist master key and the wordlist encryption IV. Before encrypting, the wordlist must be filled with blanks (0x20), so that the length of the data % 16 is 0. The client holds the IVs in memory because it needs to be transmitted to the server during the registration request.

4. Deriving the 256 bit KDF password

After encrypting the mnemonic and the wordlist with the random master keys, the client needs to derive a KDF password from the user's password. This KDF password will be used to encrypt the master keys. To derive the KDF password, the client first needs to create a random salt (from now on named KDF salt). The salt length should be 32 bytes. Next the client derives the KDF password using the generated KDF salt, the pbkdf2 derivation algorithm "PBKDF2WithHmacSHA1" with 20.000 permutations.

5. Encrypt the master keys

After deriving the KDF password, the client must encrypt the master keys with the KDF password. To encrypt the master keys, the client should use the cipher algorithm "AES/CBC/NoPadding". It first creates two new random IVs with the length of 16 bytes (from now on named master key encryption IVs). Then it encrypts the master keys with the KDF password using the created IVs. The client keeps the IVs in memory because it needs to be transmitted to the server during the registration request.

6. Remove the decrypted master keys, user's password and KDF password

After creating the data described in the steps above, the client immediately removes the decrypted master keys from its memory. It only keeps the encrypted version of the master keys in memory because it needs to be transmitted to the server during the registration request. The client also deletes the user's password, the shuffled plaintext wordlist and KDF password from its memory but keeps the KDF salt.

The client should now have following payment-relevant data that will be transmitted to the server during the registration process:

- KDF salt
- Master key encryption IVs (for both master keys)
- Encrypted master keys (both)
- Mnemonic encryption IV
- Wordlist encryption IV
- Encrypted mnemonic
- Encrypted wordlist
- Public key of index 0
- Public key of index 188

The client also still holds the user's decrypted mnemonic (the words, not indexes). It will need it later for the mnemonic quiz. It should currently NOT keep any of following:

- User's password
- KDF password
- Decrypted master keys
- Decrypted shuffled wordlist
- Any stellar account seeds

4.6.4 Server interface for the user registration

After the user has completed the registration form, the client has validated the input and prepared the payment-relevant data, he can try to register the user via the server's registration interface. The server checks the data and if everything is correct it creates the user account and saves the data in the portal's database. This chapter describes the associated server interface, its parameters and the possible return values.

Interface to register new users		
Method	**Description**	**Response**
register_user **POST** /portal/user/register_user	The client calls this interface to register the new user. **Parameters:** • email ○ mandatory ○ email address of the user: string • kdf_salt ○ mandatory ○ The salt used to derive the KDF password from the user's password: Base-64 string • mnemonic_master_key ○ mandatory ○ The **encrypted** mnemonic master key: Base-64 string • mnemonic_master_iv ○ mandatory	

    - The mnemonic master key encryption IV: Base-64 string
- wordlist_master_key
    - Mandatory
    - The wordlist master key: Base 64 string
- wordlist_master_iv
    - mandatory
    - The wordlist master key encryption IV: Base 64 String
- mnemonic
    - mandatory
    - The **encrypted** mnemonic (list of indexes): Base-64 string
- mnemonic_iv
    - mandatory
    - The mnemonic encryption IV: Base-64 string
- wordlist
    - mandatory
    - The encrypted, shuffled wordlist: Base64 string
    - The wordlist will be in form: base64(AES_encrypt(wordlist in UTF8))
    - The single words will be separated by comma
    - The wordlist will be padded with empty strings, so the client must trim all spaces before building the list
- wordlist_iv
    - mandatory
    - The wordlist encryption IV: Base-64 string
- public_key_0
    - mandatory
    - The public key at index 0: Base-64 string – already is base 64 encoded when derived
- public_key_188
    - mandatory
    - The public key at index 188: Base-64 string – already is base 64 encoded when derived
- salutation
    - configurable if optional or mandatory
    - The salutation for the user: string
    - E.g. “Mr.”
- title
    - configurable if optional or mandatory
    - The academic title of the user: string
    - E.g. “Dr.”
    -
- forename
    - configurable if optional or mandatory
    - The forename of the user: string
    - e.g. “John”
- lastname
    - configurable if optional or mandatory
    - The last name of the user: string
    - E.g. “Doe”
- company
    - configurable if optional or mandatory

	○ The company of the user: string ○ E.g. “Apple Inc.” • street_address ○ configurable if optional or mandatory ○ The street address of the user: string ○ E.g. “Bronx Park” • street_number ○ configurable if optional or mandatory ○ The street address number of the user: string ○ E.g. “291 A” • zip_code ○ configurable if optional or mandatory ○ The zip code of the user: string ○ E.g. “10468” • city ○ configurable if optional or mandatory ○ The city of the user: string ○ E.g. “New York” • state ○ mandatory for US, optional for others ○ The state of the user: string ○ E.g. “New York” • country_code ○ configurable if optional or mandatory ○ The country code of the user: string ○ E.g. “US” - (ISO 3166) • nationality ○ configurable if optional or mandatory ○ The nationality of the user: string ○ E.g. “DE” - (ISO 3166) • mobile_nr ○ configurable if optional or mandatory ○ The mobile phone number of the user: string ○ E.g. “+4915142332125” • birth_day ○ configurable if optional or mandatory ○ The birthday of the user: string ○ E.g. “1985-12-20” – ISO 8601 – YYYY-MM-DD • birth_place ○ configurable if optional or mandatory ○ The birth place of the user: string ○ E.g. “München”	

Data Objects	
Name	**JSON**

RegisterUserResponse
On status code 200
	{
	/* optional, 2FA secret, only if status is success*/
	“tfa_secret” : “9187737337”,
	
	/* optional, 2FA secret QR Image, only if status is success – base64 encoded - PNG*/
	“tfa_qr_image” : “BAH192BBS…”
	}

ValidationError **On status code 400 or 500**	{ “error_code”: 1000, /* mandatory, possible values: see table below */

	/* name of the parameter, see above */ "parameter_name": "mobile_nr", /* message for the client/user, if needed */ "user_error_message_key": "register_user.mobile_nr.wrong_country" /* optional, message for the developer from server*/ "error_message": "Mobile number does not match country of residence" }
Possible error codes	
Error code	**Description**
1000	Invalid argument from a user input. If there is something wrong with one of the values given by the user.
1001	Email-address already exists in the database. An account could not be created.
1002	Missing mandatory parameter.
1003	Reserved for future use.
1004	Invalid length of parameter value.
1005	Master key encryption IV is the same as mnemonic encryption IV.
JWT Token	
Depends on status code and success.	The server only returns a JWT token (partially authenticated) on status code 200 if the new account was successfully created

Table 13 – server interface for registration form

As soon as the server receives the request, it immediately validates the data. If any of the values is not valid, the server returns an input validation error naming the field that is not valid. After validating the values, the server checks if a user with the same email-address already exists in the portal's database. If this is the case, then no new user account can be created. The server responds to the client with the appropriate hint. The client can now inform the user and give him the possibility to specify another email address or to log in with the existing email address. The server does not return a single error. It collects all errors and then returns a list to the client so that the client / user can handle them all at once.

If there were no errors, then the server also transmits a JWT token (partially authenticated) to the client. The client will need this JWT token during the 2FA registration process.

If there were errors, then the server returns a list of found errors as a value for the JSON field and sets the HTTP-Status 400.

The server will perform the following validations:

- All mandatory parameters have been transmitted
- All validations described in the chapter Registration form
- Mobile number must match country of residence
- length of KDF password salt is (32 bytes – base 64 decoded).
- length of master key encryption IVs are (16 bytes – base 64 decoded).
- Correct length of mnemonic encryption IV and wordlist encryption IV are (16 bytes – base 64 decoded).
- all IVs are different
- length of public key at index 0 is 56 characters
- length of public key at index 188 is 56 characters

- length of encrypted master keys are (32 bytes – base 64 decoded).
- length of encrypted mnemonic – list of indexes referring to words in the shuffled wordlist is (48 bytes – base64 decode, because each index in the list has 2 bytes)
- length of the encrypted wordlist

After successfully validating the data, the server creates the user and stores its data in the database of the portal. As already described above, the registration process is only completed after the user's account has been created, the user has completed the 2FA registration, has confirmed his email-address and his mnemonic. After the successful creation of the user, the server sends the data relevant for 2FA registration to the client together with a JWT token (partially authenticated). The following diagram shows an overview of the implementation of this first part of the registration.

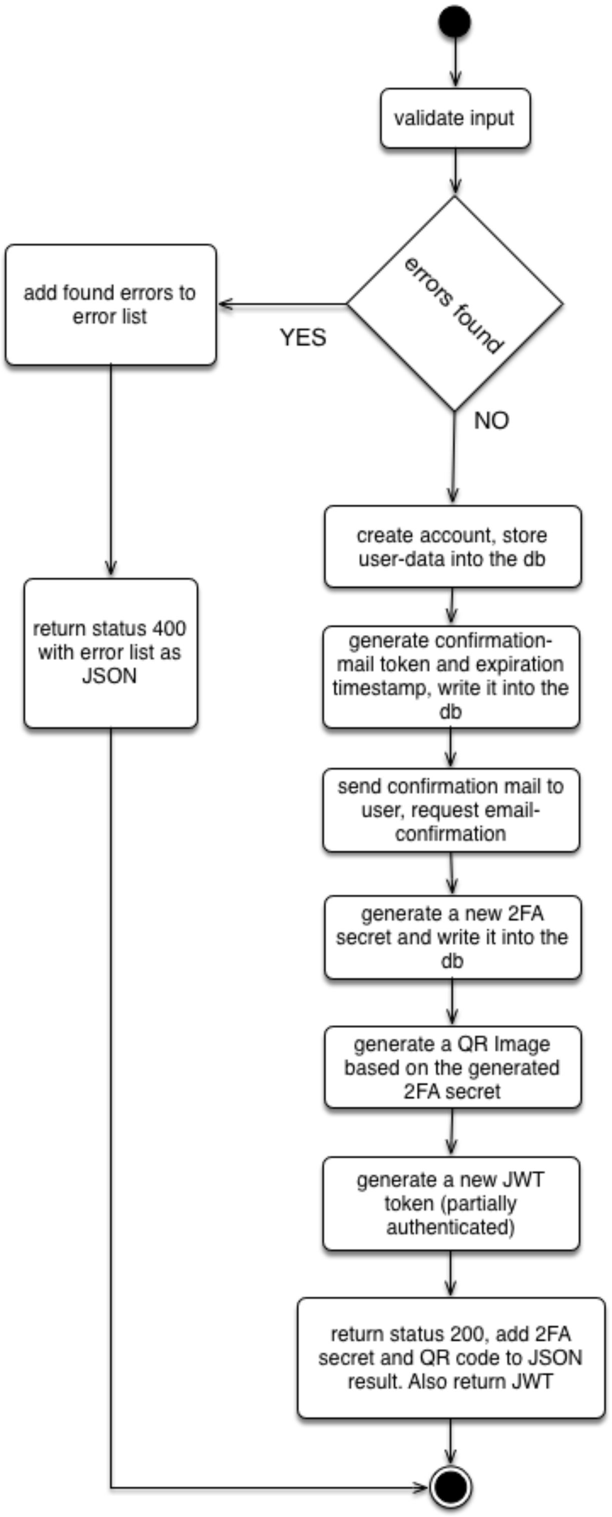

validate input
errors found
YES
NO
add found errors to error list
return status 400 with error list as JSON
create account, store user-data into the db
generate confirmation-mail token and expiration timestamp, write it into the db
send confirmation mail to user, request email-confirmation
generate a new 2FA secret and write it into the db
generate a QR Image based on the generated 2FA secret
generate a new JWT token (partially authenticated)
return status 200, add 2FA secret and QR code to JSON result. Also return JWT

Image 6 – Implementation overview: server-interface user registration

4.6.5 Email-Confirmation on registration

During the registration process, the server sends a mail to the user for the user to confirm his email address. This mail includes a link with a generated mail-token. When the user clicks on the link, he gets to a web browser, a client that forwards the mail token to the server. The client uses the functionality described in chapter Confirm token by mail for validating the token.

First the server checks if the parameter "token" is filled and if it is a valid token, the server next looks for the user in the database to which the token belongs. If he finds exactly one user, the token could be associated successfully. Otherwise, the server will return an error and the client can show a page, where user can request a new confirmation email with a new token.

If the token is valid, then the server sets the mail confirmed flag in the user's database entry to true and returns status 200 to the client with a partially authenticated JWT token.

Following diagram shows an overview of the described process:

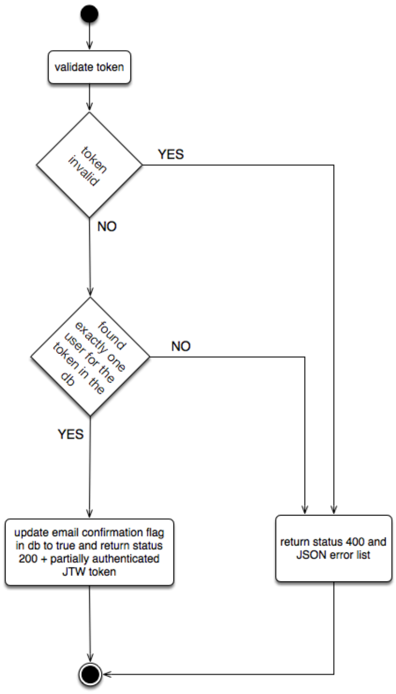

validate token
token invalid
YES
NO
found exactly one user for the token in the db
NO
YES
update email confirmation flag in db to true and return status 200 + partially authenticated JTW token
return status 400 and JSON error list

Image 7 – Implementation overview: server-interface for email confirmation

If the user reaches the error page because of invalid token, she can request a new confirmation mail. To do this, she must enter his email address and press the resend button. The client calls the following server interface:

Interface to resend confirmation mail		
Method	**Description**	**Response**
resend_confirmation_mail **POST** /portal/user/ resend_confirmation_mail	User requests the resending of the confirmation email-mail. This can occur for example if the token from the previous mail already expired. **Parameters:** • email	**ResendConfirmationMailResponse**

	○ mandatory ○ the email-address of the user to resend the confirmation mail.	

Data Objects	
Name	**JSON**

ResendConfirmation MailResponse
On status 200
	{
	/* empty message */
	}

ValidationError
On status 400
	{
	“error_code”: 1000, /* mandatory, possible values: see table below */
	/* name of the parameter, see above */
	“parameter_name”: “email”,
	/* message for the client/user, if needed */
	“user_error_message_key”: “confirm_mail.email.notFound”
	/* optional, message for the developer from server*/
	“error_message”: “Email-address not found in database.”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. If the email has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.
1008	Email-address already confirmed.

Table 14 – server interface for resending confirmation mail

As soon as the server receives the request, it checks first whether the given email address has a valid format (see also chapter Registration form). If it is an invalid formatted address, the server returns an error with code 1000 (invalid argument).

Next, the server checks if there is a user in the database that has the specified email address. If this is not the case, the server also returns an error with code 1000 and corresponding error message to the client.

If the server finds the associated user, the server next checks whether the user's email address has already been confirmed. If this is the case, the server returns an error code 1008 for the field email.

If the email address of the user is not yet confirmed, the server sends a new confirmation email with the new token to the user.

Following diagram shows an overview of the described process:

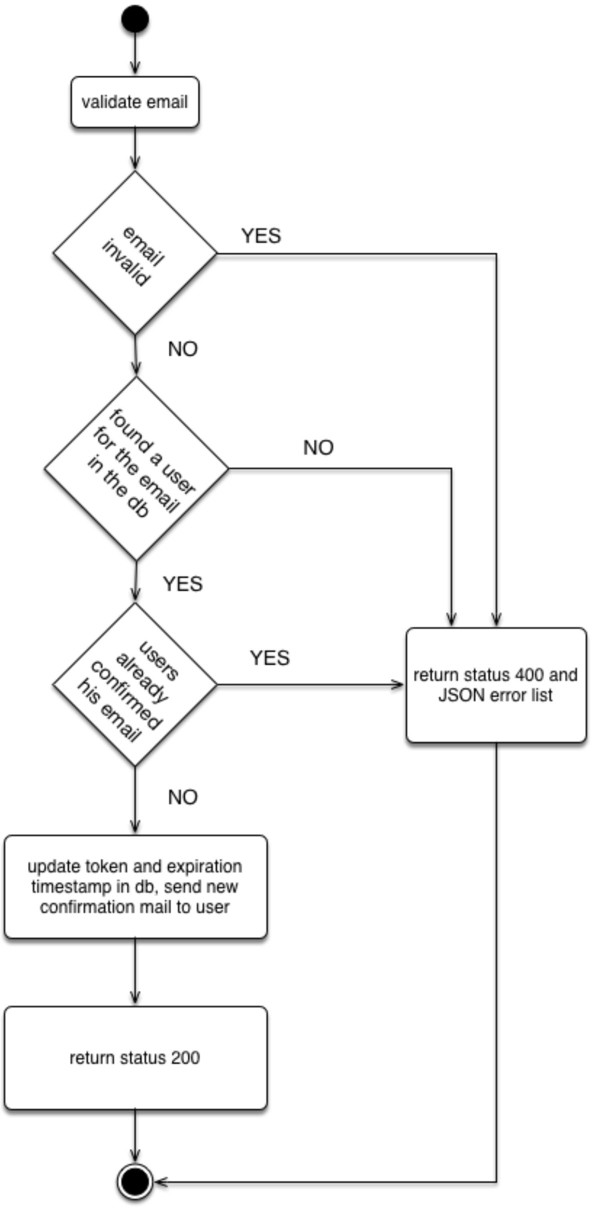

validate email
email invalid
YES
NO
found a user for the email in the db
NO
YES
users already confirmed his email
YES
return status 400 and JSON error list
NO
update token and expiration timestamp in db, send new confirmation mail to user
return status 200

Image 8 – Implementation overview: server-interface to resend email-confirmation

4.6.6 Performing the 2FA registration

In the first registration step, after the server has created the user account, it returns to the client the user's 2FA secret and the corresponding QR image. These are displayed by the client, so that the user can perform his 2FA registration. The user scans the QR code, for example with the "Google Authenticator" app or enters the 2FA secret there directly. The "Google Authenticator" app now generates a new 2FA code every 30 seconds. The user must next enter the current 2FA code in the client to complete his 2FA registration. As soon as the user enters the current 2FA code, the client sends it together with the previously received JWT token to the server.

For this, the server provides the following interface:

Interface to confirm 2FA registration		
Method	**Description**	**Response**
confirm_tfa_regsitration **POST** /portal/user/auth/ confirm_tfa_registration	After scanning the QR-image or directly inserting the 2FA secret into an authenticator app e.g. "Google Authenticator", the user confirms the 2FA registration by inserting the current 2FA code. **Parameters:** • tfa_code ○ mandatory ○ the current 2FA code • JWT Token (partially authenticated) ○ mandatory ○ The JWT token that the client received during the first user registration step or at login.	**2FARegistrationResponse**
Data Objects		
Name	**JSON**	

2FARegistrationResponse on status 200	{
	"mail_confirmed": bool,
	"tfa_confirmed": bool,
	"menmonic_confirmed": bool
	}	

ValidationError
On status 400 or 500
	{
	
	"error_code": 1000, /* mandatory, possible values: see table below */
	
	/* name of the parameter, see above */
	"parameter_name": "tfa_code",
	
	/* message for the client/user, if needed */
	"user_error_message_key": "confirm_tfa_regsitration.tfa_code.invalid"
	
	/* optional, message for the developer from server*/
	"error_message": "2FA code is invalid."
	}	

Possible errors		
Error code	**Description**	
1000	Invalid argument. If the email or JWT token has a wrong format or could not be found in the database.	
1002	Missing mandatory parameter.	
1007	JWT Token expired.	
1009	2FA already confirmed.	

Table 15 – server interface for confirming the 2FA registration

As soon as the server receives the 2FA code and JWT token (partially authenticated) via the interface mentioned above, he first checks the JWT token. This can be invalid (not findable) or expired.

If the JWT token is valid, the next thing to check is whether the user has already completed the 2FA registration. This can be the case if the user has left the browser window with the 2FA registration open and has already completed the registration in another browser window (see also chapter Login). If the user then returns to the first browser window and tries to complete the registration again, he is already registered and the server will return the error code 1009 for the field “tfa_code” to the client.

If the 2FA registration has not yet been completed, the received 2FA code will be checked next. If the 2FA code is incorrect (e.g. the user has made a typing mistake), the server responds with error code 1000 for the field “tfa_code”.

If the 2FA code entered by the user is valid, the server sets the corresponding 2FA-registration-confirmed flag in the database entry of the user to true and sends back to the client a fully authenticated JWT token. The client can now confirm the user the successful 2FA registration and forward the user to the dashboard.

Following diagram shows a rough server-side overview of the described process:

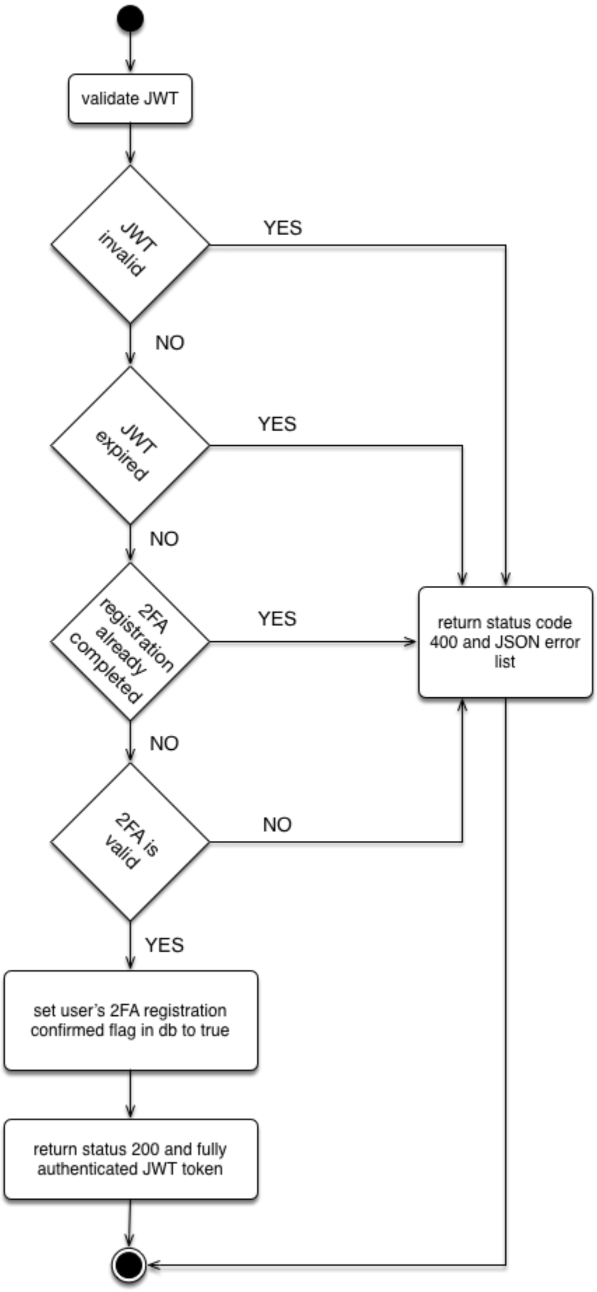

validate JWT
JWT invalid
YES
NO
JWT expired
YES
NO
2FA registration already completed
YES
NO
2FA is valid
NO
YES
return status code 400 and JSON error list
set user’s 2FA registration confirmed flag in db to true
return status 200 and fully authenticated JWT token

Image 9 – Implementation overview: server-interface to confirm 2FA registration

4.6.7 Requesting registration status

The client can query the user's registration status from the server. The server responds whether the user:

- has completed the 2FA registration
- has confirmed his email address
- successfully written down the mnemonic and confirmed it by filling the quiz

This query is currently required at the following locations:

- after 2FA registration, the client must know whether the user has already confirmed his email address
- after login, the client must know whether the user has already confirmed his email address
- after login, the client must know whether the user has already written down his mnemonic and confirmed it

This function can be called by 2 API endpoints.

1) “/portal/user/auth/get_user_registration_status” if the user is already partially authenticated (JWT partially authenticated present)
2) “/portal/user/dashboard/ get_user_registration_status” if the user is already fully authenticated (JWT fully authenticated present)

Interface for the API calls		
Method	**Description**	**Response**
get_user_registration_status **GET** /portal/auth/ get_user_*registration*_status	This interface is used for requesting the user registration status, that is needed for the frontend flow. **Parameters:** ○ JWT token (partially authenticated)	**UserRegistrationStatus**
get_user_details **GET** /ico/dashboard/ get_user_*registration*_status	**Parameters:** ○ JWT Token (fully authenticated)	**UserRegistrationStatus**
Data Objects		
Name	**JSON**	
UserRegistrationStatus on status 200	{ “mail_confirmed”: bool,	

	“tfa_confirmed”: bool, “menmonic_confirmed”: bool }

ValidationError
On status 400 or 500
	{
	}

Possible errors	
Error code	**Description**

Table 16 – server interface for the retrieving the user details

4.6.8 Registration flow in the client

Following activity diagram shows the flow of the registration process in the client.

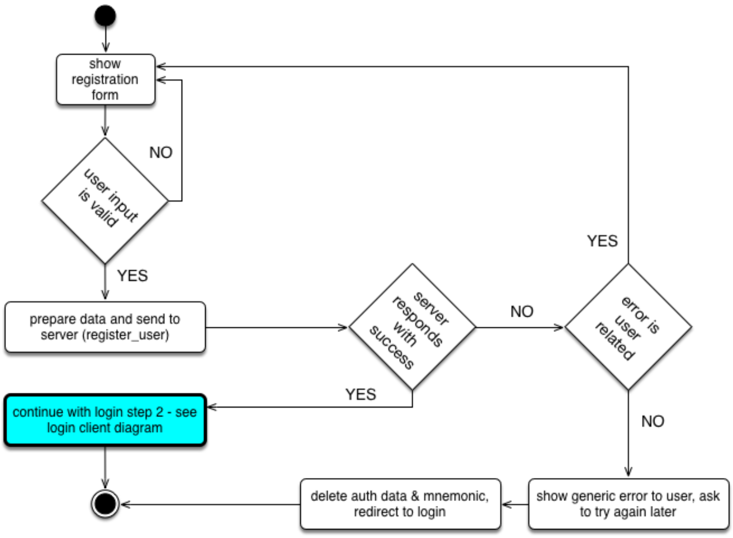

show registration form
user input is valid
NO
YES
prepare data and send to server (register_user)
server responds with success
NO
YES
error is user related
YES
NO
continue with login step 2 - see login client diagram
delete auth data & mnemonic, redirect to login
show generic error to user, ask to try again later

Image 10 – Implementation overview: registration flow in the client

4.7 Login

As described in the beginning, the login process consists of two steps. However, these steps are not visible to the user. The user only must enter his email address, password and current 2FA code. However, it may be that the user cannot enter a current 2FA code, because he has not yet completed the 2FA registration (for example, because he has closed the browser window during the 2FA registration). Therefore, the input field 2FA Code must be an optional field in the login form.

This gives rise to two possibilities that must be dealt with separately:

1. The user enters email address, password and current 2FA code

2. The user enters only email address and password

In the following chapters, both cases are treated separately.

4.7.1 User provides 2FA Code at login

In this case, the first step of the login process is that the client sends the email address of the user along with the inserted 2FA code to the server. Following interface is used by the client to do so:

Interface for first login step		
Method	**Description**	**Response**
login_step1 **POST** /portal/user/ login_step1	This interface is used for the first login step. **Parameters:** • email ○ mandatory ○ the users email-address • tfa_code ○ mandatory if user is 2FA registered ○ The current 2FA code.	**LoginStep1Response**
Data Objects		
Name	**JSON**	

LoginStep1Response	{
	“login_step1_status”: “success”, /* possible values: “success” , “error”*/
	“kdf_password_salt”: “MSJSUZDFD33…”, // base64
	“encrypted_mnemonic_master_key”: “ASAUZDFD672…”, // base64
	“mnemonic_master_key_encryption_iv”: “HAHA773JJ…”, // base64
	“encrypted_mnemonic”: “JANAAS82…”, // base64
	“mnemonic_encryption_iv”: “HHSSA922…”, // base64
	“encrypted_wordlist_master_key”: “KAJJASS12…”,
	“wordlist_master_key_encryption_iv” : “AOOSPS..”,
	“encrypted_wordlist” : ”AKSKKSKSK….”,
	“wordlist_encryption_iv”: string,
	“public_key_index0”: “AMAMS992…”, // base64
	}	

ValidationError	{	

	“error_code”: 1002, /* mandatory, possible values: see table below */ /* name of the parameter, see above */ “parameter_name”: “tfa_code”, /* message for the client/user, if needed */ “user_error_message_key”: “login_step1.tfa_code.missing” /* optional, message for the developer from server*/ “error_message”: “2FA code is missing.” }
Possible errors	
Error code	**Description**
1000	Invalid argument. If the email or 2FA code has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.
JWT Token	
Depends on success.	The server only returns a JWT token (partially authenticated) if the user could partially be logged in and no error occurred.

Table 17 – server interface for the first login step

The server checks the data and if it is valid, it partially logs in the user and returns a JWT token (partially authenticated), the KDF password salt, the encrypted master key, the master key encryption IV, the encrypted mnemonic and the mnemonic encryption IV and public key at index 0 of the user. This data is required by the client to complete the next step of the login process.

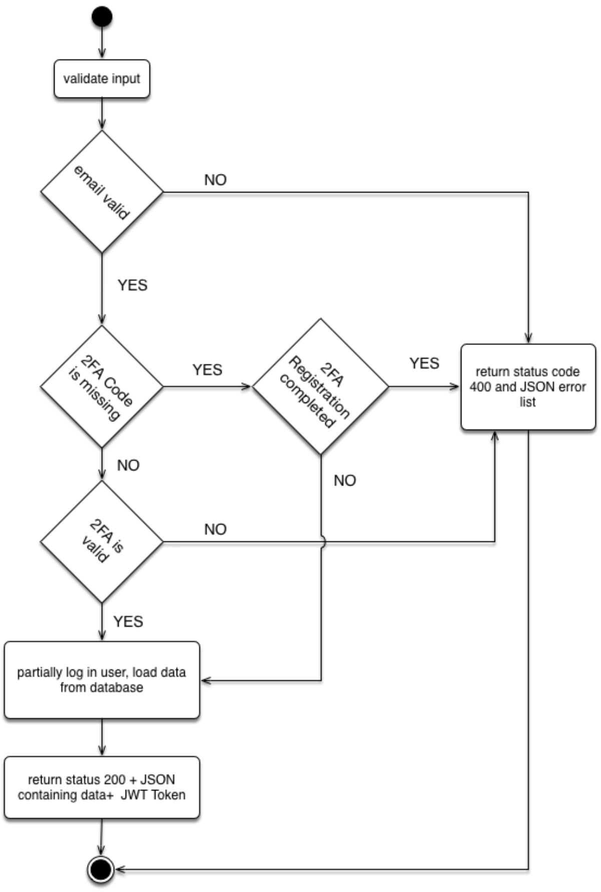

validate input
email valid
NO
YES
2FA Code is missing
YES
2FA Registration completed
YES
return status code 400 and JSON error list
NO
NO
2FA is valid
NO
YES
partially log in user, load data from database
return status 200 + JSON containing data+ JWT Token

Image 11 – Implementation overview: partially log in user with 2FA code

In the second step, the client first checks the password of the user. To do this, the client performs the following steps:

1. Derivation of the KDF password from the user's password and KDF salt.

2. Decryption of the master keys with the KDF password and master key encryption IVs

3. Decryption of the mnemonic (list of indexes from wordlist) with the mnemonic master key and mnemonic encryption IV. If the list does not contain only numbers, this already is a proof that the user inserted the wrong password. The client will not continue in that case but ask the user to insert his password again

4. Decryption of the shuffled wordlist with the wordlist master key and wordlist encryption IV

5. Building up the mnemonic (24 words) by selecting the words from the shuffled wordlist based on the list of indexes. If the indexes cannot be found in the wordlist, this is an error that we cannot fix and the client will ask the user to insert his password again. By doing so, the user will repeat inserting his password resulting into the same error until the user will press the “forgot password” link and reset his password which will lead into resolving this problem because we then will override the encrypted mnemonic and wordlist.

6. Derivation of the public key of index 0 from the mnemonic (24 words)

7. Comparison of the derived public key of index 0 with the public key of index 0 obtained from the server (they must be equal)

If step 3. or step 7. is not correct, the user entered the wrong password. If step 5 is not correct, an error occurred. The client then asks the user to re-enter his password. If the user enters the password again, the client repeats the second step of the login process. If step 3. and step 5. and step 7. are correct, the user entered the correct password.

The client now knows that the user has entered the correct password. However, he still must tell the server and prove it. For this, the client first derives the public key of index 188 from the mnemonic and sends it to the server together with the JWT token (partially authenticated) obtained in step one. It uses following server interface to do so:

Interface for second login step		
Method	**Description**	**Response**
login_step2 **POST** /portal/user/auth/login_step2	This interface is used for the second login step. **Parameters:** • JWT Token (partially authenticated) ○ mandatory ○ The JWT Token obtained in step 1 • key ○ mandatory ○ public key at index 188 derived from user’s mnemonic	**LoginStep2Response**
Data Objects		

Name	JSON

LoginStep2Response	{
	/* optional, only if status is “2FA registration required” */
	“tfa_secret” : “9187737337”,
	/* optional, only if status is “2FA registration required” */
	“tfa_qr_image” : “BAH192BBS…”
	/* optional, only if user did not complete the mnemonic quiz, only “true” is a possible value. */
	“mail_confirmed”: bool,
	“tfa_confirmed”: bool,
	“mnemonic_confirmed”: bool
	}

ValidationError	{
	“error_code”: 1000, /* mandatory, possible values: see table below */
	/* name of the parameter, see above */
	“parameter_name”: “key”,
	/* message for the client/user, if needed */
	“user_error_message_key”: “loginStep2.key.invalid”
	/* optional, message for the developer from server*/
	“error_message”: “Cannot login user, public key is invalid.”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. Public key at index 188 is not valid or JWT token is not valid.
1002	Missing mandatory parameter.
1007	JWT token expired.

Table 18 – server interface for the second login step

The server first checks the received JWT token (partially authenticated). The token may be invalid (wrong format or not present) or it may have expired. If this is the case, the server responds with an associated error code. If the JWT token is valid, the server next checks the received public key of index 188. If this is not correct, the server responds with a corresponding error code. If the public key is correct, it logs in the user completely and communicates the successful login to the client. It returns the fully authenticated JWT Token to the client.

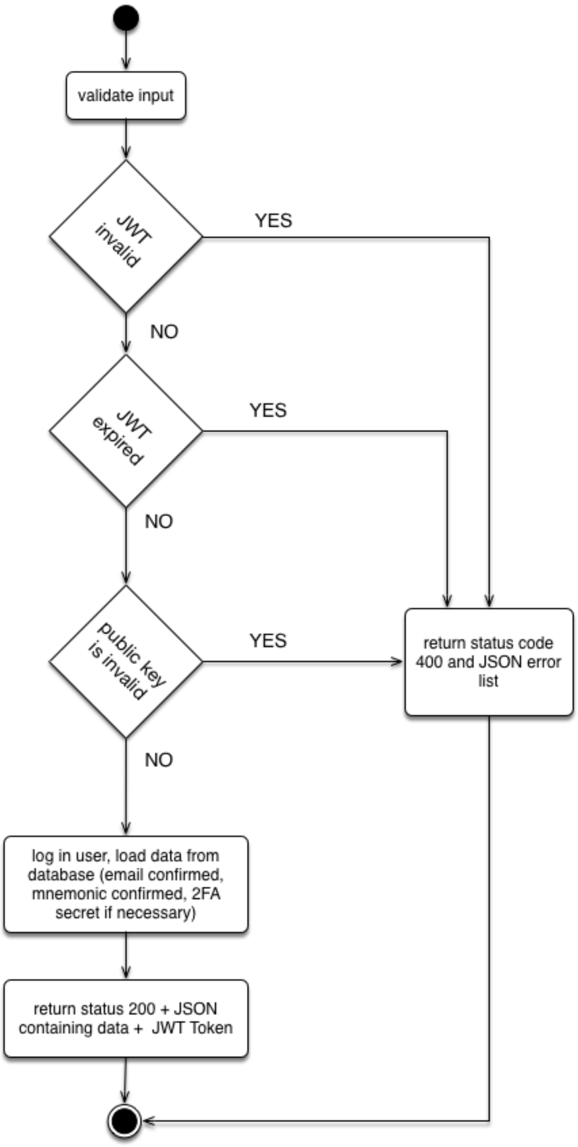

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
Public key is invalid
YES
NO
return status code 400 and JSON error list
log in user, load data from database (email confirmed, mnemonic confirmed, 2FA secret if necessary)
return status 200 + JSON containing data + JWT Token

Image 12 – Implementation overview: completely log in user with 2FA code

4.7.2 User does not provide 2FA Code at login

In this case only email address and password will be checked. If these are valid, the user is prompted to perform the 2FA registration. He only gets to the dashboard as soon as he has completed the 2FA registration, confirmed his email address and mnemonic.

In the first step, the client sends only the user's email address to the server. It uses the server interface “login_step1” as already described in the chapter User provides 2FA code at login.

The server then checks whether the email address exists and whether in fact no 2FA registration has yet been made. If the 2FA registration has already been made, the server responds with an associated error code and the client must now force the user to enter his 2FA code. If no 2FA registration has been made, the server responds with a JWT token, the KDF salt, the encrypted master key, the master key encryption IV, the encrypted mnemonic and the mnemonic encryption IV and public key at index 0 of the user.

In the second step, the client first checks the password of the user. To do this, the client performs the following steps:

1. Derivation of the KDF password from the user's password and KDF salt.

2. Decryption of the master keys with the KDF password and master key encryption IVs

3. Decryption of the mnemonic (list of indexes from wordlist) with the mnemonic master key and mnemonic encryption IV. If the list does not contain only numbers, this already is a proof that the user inserted the wrong password. The client will not continue in that case but ask the user to insert his password again

4. Decryption of the wordlist with the wordlist master key and wordlist encryption IV

5. Building up the mnemonic (24 words) by selecting the words from the wordlist based on the list of indexes. If the indexes cannot be found in the wordlist, this is an error that we cannot fix and the client will ask the user to insert his password again. By doing so, the will repeat inserting his password resulting into the same error until the user will press the "forgot password" link and reset his password which will lead into resolving this problem because we then will override the encrypted mnemonic.

6. Derivation of the public key of index 0 from the mnemonic (24 words)

7. Comparison of the derived public key of index 0 with the public key of index 0 obtained from the server (they must be equal)

If step 3. or step 7. is not correct, the user entered the wrong password. If step 5 is not correct, an error occurred. The client then asks the user to re-enter his password. If the user enters the password again, the client repeats the second step of the login process.

The client now knows that the user has entered the correct password. However, he still must tell the server and prove it. For this, the client first derives the public key of index 188 from the mnemonic and sends it to the server together with the JWT token (partially authenticated) obtained in step one. To do so it uses the server interface "login_step2" as described in the chapter User provides 2FA code at login.

The server first checks the received JWT token (partially authenticated). The token may be invalid or it may have expired. If this is the case, the server responds with an associated error code. If the JWT token is valid, the server next checks the received public key of index 188. If this is not correct, the server responds with a corresponding error code. If the public key of index 188 is correct, the server responds with success status 200 and with the 2FA secret and associated QR image. The client displays these so that the user can perform the 2FA registration process. After the user has entered the current 2FA code to complete the 2FA registration, this is sent by the client together with the JWT token (partially authenticated) to the server. The client uses following server interface defined in the chapter Performing the 2FA registration.

The server checks the value of the 2FA code. If the 2FA code is invalid, the server responds with a corresponding error code and the client can request re-input of the code from the user. If the 2FA code is correct, the server logs in the user completely and informs the client about it (sends fully authenticated JWT to the client). The client can now show the user his dashboard (only if email address and mnemonic are confirmed).

4.7.3 Request 2FA Secret in mobile app after login

To ensure a good usability of the mobile apps, the mobile app should store the user's 2FA secret internally in its keychain after the first login of the user, so that the user does not have to insert his 2FA code again the next time the login is called. The 2FA code should be generated internally by the app and passed to the server for login. The user should only have to re-enter his password. Should the user explicitly log out, the 2FA secret should be deleted from the keychain again.

To get the 2FA secret, the app uses the following server interface:

Interface to request the 2FA secret after login		
Method	**Description**	**Response**
tfa_secret **POST** /portal/dashboard/tfa_secret	The mobile app requests the 2FA secret of the user immediately after first login. **Parameters:** • JWT token (fully authenticated) ○ Mandatory • Key ○ Public key of index 188	**TfaSecret**
Data Objects		
Name	**JSON**	

TfaSecret
On status 200
	{
	tfa_secret = “87872837218”
	}	

ValidationError
On Status 400 or 500
	{
	“error_code”: 1013,
	“parameter_name”: “key”,
	“user_error_message_key”: “”
	“error_message”: “Invalid public key”
	}	

Possible errors		
Error code	**Description**	
1002	Missing mandatory parameter.	
1013	Invalid public key 188	

Table 19 – server interface to request the 2fa secret

The server first validates the JWT Token (fully authenticated) and the public key at index 188. If they are invalid it returns the corresponding error. If they are correct, the server returns the 2FA secret.

After obtaining the 2FA secret from the server, the client should immediately delete the public key of index 188.

4.7.4 Login flow in the client

Following activity diagram shows the flow of the login process in the client.

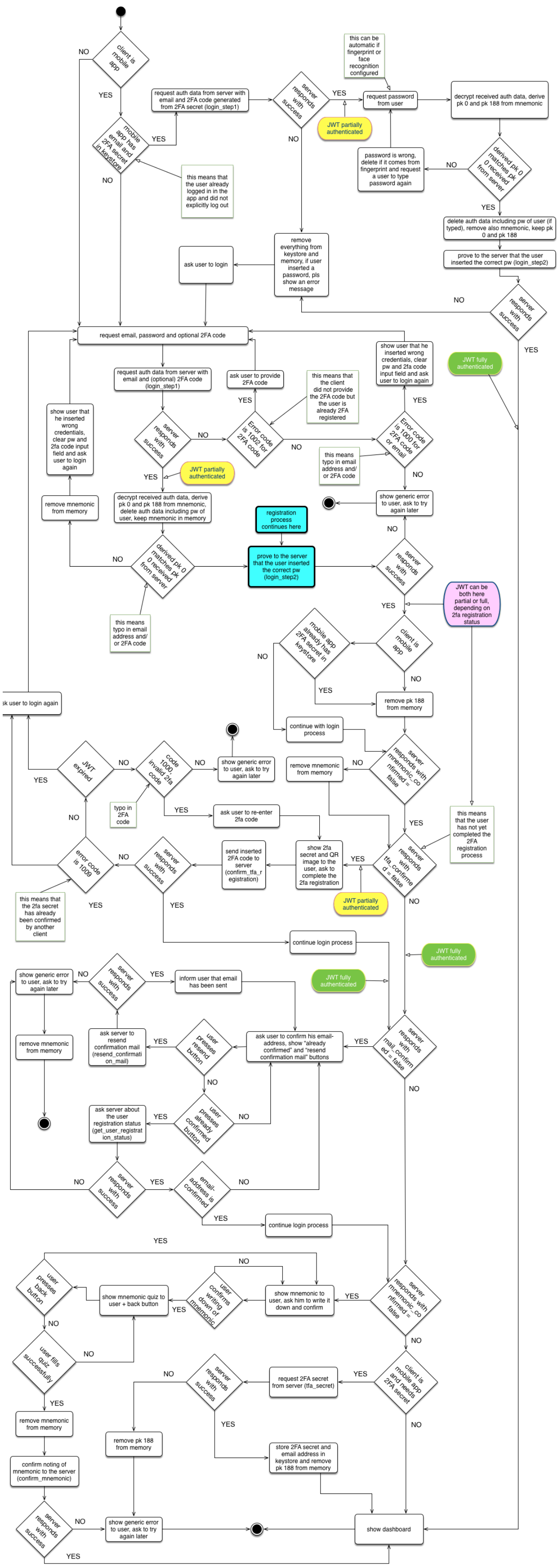

client is mobile app
mobile app has email and 2FA secret in keystore
request auth data from server with email and 2FA code generated from 2FA secret (login_step1)
server responds with success
this can be automatic if fingerprint or face recognition configured
request password from user
JWT partially authenticated
decrypt received auth data, derive pk 0 and pk 188 from mnemonic
this means that the user already logged in in the app and did not explicitly log out
password is wrong, delete if it comes from fingerprint and request a user to type password again
derived pk 0 matches pk 0 received from server
delete auth data including pw of user (if typed), remove also mnemonic, keep pk 0 and pk 188
prove to the server that the user inserted the correct pw (login_step2)
remove everything from keystore and memory, if user inserted a password, pls show an error message
ask user to login
request email, password and optional 2FA code
request auth data from server with email and (optional) 2FA code (login_step1)
ask user to provide 2FA code
this means that the client did not provide the 2FA code but the user is already 2FA registered
show user that he inserted wrong credentials, clear pw and 2fa code input field and ask user to login again
JWT fully authenticated
show user that he inserted wrong credentials, clear pw and 2fa code input field and ask user to login again
Error code is 1002 for 2FA code
Error code is 1000 for 2FA code or email
this means typo in email address and/ or 2FA code
decrypt received auth data, derive pk 0 and pk 188 from mnemonic, delete auth data including pw of user, keep mnemonic in memory
remove mnemonic from memory
show generic error to user, ask to try again later
registration process continues here
prove to the server that the user inserted the correct pw (login_step2)
this means typo in email address and/ or 2FA code
JWT can be both here partial or full, depending on 2fa registration status
mobile app already has 2FA secret in keystore
remove pk 188 from memory
continue with login process
sk user to login again
JWT expired
code 1000, invalid 2fa code
show generic error to user, ask to try again later
remove mnemonic from memory
server responds with mnemonic_confirmed = false
typo in 2FA code
ask user to re-enter 2fa code
this means that the user has not yet completed the 2FA registration process
error code is 1009
send inserted 2FA code to server (confirm_tfa_registration)
show 2fa secret and QR image to the user, ask to complete the 2fa registration
server responds with tfa_confirmed = false
this means that the 2fa secret has already been confirmed by another client
continue login process
show generic error to user, ask to try again later
inform user that email has been sent
remove mnemonic from memory
ask server to resend confirmation mail (resend_confirmation_mail)
user presses resend button
ask user to confirm his email-address, show “already confirmed” and “resend confirmation mail” buttons
server responds with mail_confirmed = false
ask server about the user registration status (get_user_registration_status)
user presses already confirmed button
email-address is confirmed
continue login process
user presses back button
show mnemonic quiz to user + back button
user confirms writing down of mnemonic
show mnemonic to user, ask him to write it down and confirm
server responds with mnemonic_confirmed = false
user fills quiz successfully
request 2FA secret from server (tfa_secret)
client is mobile app and needs 2FA secret
remove mnemonic from memory
remove pk 188 from memory
store 2FA secret and email address in keystore and remove pk 188 from memory
confirm noting of mnemonic to the server (confirm_mnemonic)
show generic error to user, ask to try again later
show dashboard
YES
NO

Image 13 – Implementation overview: login flow in the client

4.8 Confirm token by mail

There are some flows inside the portal, where the user must confirm, that he has access to the specified email address. This interface is used at different stages and is implemented in one server API call.
This interface is called from different endpoints inside the client.

Interface to confirm a mail token		
Method	**Description**	**Response**
confirm_token **POST** /portal/user/confirm_token	The user received an email with a token and wants to confirm this token. **Parameters:** • token ○ mandatory ○ the token from the email.	**ConfirmTokenResponse**
Data Objects		
Name	**JSON**	

ConfirmTokenResponse
On status 200
	{
	}	

ValidationError
On Status 400 or 500
	{
	“error_code”: 1006,
	“parameter_name”: “token”,
	“user_error_message_key”: “confirm_token.token.expired”
	“error_message”: “Email confirmation token expired”
	}	

Possible errors		
Error code	**Description**	
1000	Invalid argument. Token could not be found in the database.	
1002	Missing mandatory parameter.	
1006	Token expired	
1008	Email address already confirmed	

Table 20 – server interface for confirm token

The server checks, if the token can be found. If not it will return status 400 and the error code 1000. The server also checks, if the token is expired. If so, it will return status 400 and error code 1006.

If all checks passed, the server will:

- reset the confirmation key and confirmation expiry in the database
- sets the user email-address as confirmed in the database (if not already done)
- return HTTP status code 200. The response will include a JWT token header for partially authentication.

4.9 Confirm mnemonic

As soon as the user has completed the 2FA registration and has confirmed his email address successfully, the client must display the mnemonic to the user. The user must write the mnemonic down and confirm to the client that he has written it down. As soon as he has done this, the client shows a mnemonic quiz to validate the confirmation. The client shows the user 4 random words from the mnemonic and asks the user to indicate their position within the mnemonic. The first word starts at position one.

If the user enters the positions correctly, the client can assume that the user has written down his mnemonic. This must be communicated to the server next. Since the client always has a fully authenticated JWT token at the time of the query, no further proof is necessary. The client calls the following interface.

Interface to confirm the writing down of the mnemonic		
Method	**Description**	**Response**
confirm_mnemonic **POST** /portal/user/ dashboard/confirm_mnemonic	After the user filled the mnemonic quiz correctly, the client calls this interface to tell the server about the confirmation that the user has noted the mnemonic successfully. **Parameters:** • JWT token (fully authenticated authentication)	**ConfirmMnemonic Response**
Data Objects		
Name	**JSON**	

ConfirmMnemonicResponse
On status 200
	{
	
	}	

ValidationError
On Status 400 or 500
	{
	“error_code”: 1010,
	“error_message”: “Email address not confirmed”
	}	

Possible errors		
Error code	**Description**	
1010	Email address not confirmed	

Table 21 – server interface for confirming the writing down of the mnemonic

The server checks first if the JWT token is valid and then whether the user has already confirmed his email address. If this is not the case, he returns a corresponding error. If the JWT token is valid and the email address is confirmed, then the server makes an update in the database, sets the mnemonic confirmed flag to true and returns status 200 to the client.

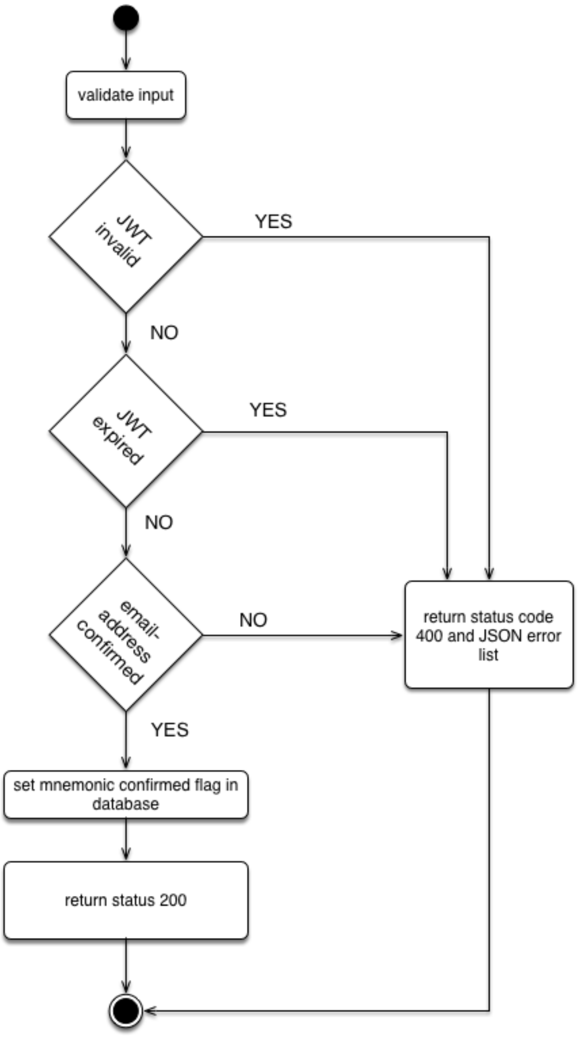


Image 14 – Implementation overview: confirm mnemonic

Next the client can show the dashboard to the user.

4.10 Password lost

If the user forgot his password, he can request the setting of a new password. To do this, he presses the "Lost password" button in the client and enters his email address. The client sends the entered email address to the server, which validates it. The server interface to be used by the client is shown below:

Interface to request the setting of a new password		
Method	**Description**	**Response**
lost_password **POST**	The user lost his password and wants to set a new password. He first inserts his email-address. The client sends the inserted email-address to the server by using this	**LostPasswordResponse**

/portal/user/lost_password	interface. **Parameters:** • email ○ mandatory ○ the email-address of the user.	

Data Objects	
Name	**JSON**

LostPasswordResponse
On status 200
	{
	}

ValidationError
On Status 400 or 500
	{
	“error_code”: 1000, /* mandatory, possible values: see table below */
	/* name of the parameter, see above */
	“parameter_name”: “email”,
	/* message for the client/user, if needed */
	“user_error_message_key”: “lost_password.email.notFound”
	/* optional, message for the developer from server*/
	“error_message”: “Email-address not found in database.”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. If the email has a wrong format or could not be found.
1002	Missing mandatory parameter.
1010	Email address is not confirmed
1011	Mnemonic is not confirmed

Table 22 – server interface for password lost step 1

If a user exists with the specified confirmed email address and confirmed mnemonic, the server sends an email to the user with the request to press the link contained therein to set a new password. If the email address of the user is not confirmed, the server responds with an error and lets the client know that email-confirmation is needed first.

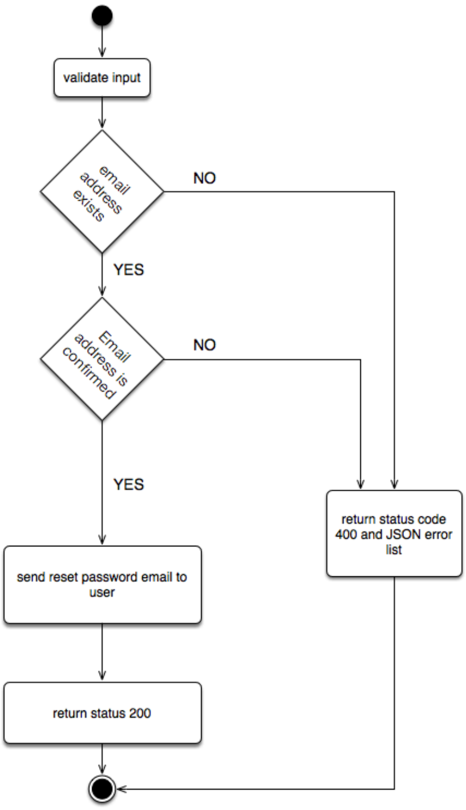

validate input
email address exists
NO
YES
Email address is confirmed
NO
YES
send reset password email to user
return status code 400 and JSON error list
return status 200

Image 15 – Implementation overview: password lost step 1

If the server sent the reset password email and the user presses the reset password link in the received mail, a new web-client forwards the token from the link in the mail to the server. The server checks the token, thus ensuring that the user has access to the specified email address. If the token is invalid or expired, the server returns the corresponding error code to the client. If the token is correct, the server returns a JWT token (partially authenticated) to the client. This API endpoint is described in the chapter Confirm token by mail.

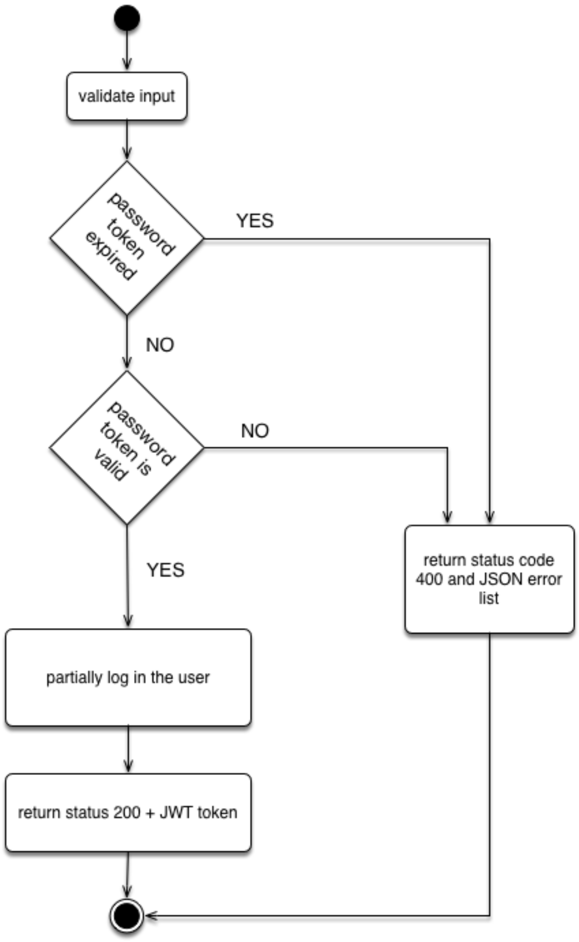

validate input
password token expired
YES
NO
password token is valid
NO
YES
return status code 400 and JSON error list
partially log in the user
return status 200 + JWT token

Image 16 – Implementation overview: password lost step 2

The user is now partially authenticated and must next enter the current 2FA code in the client. If this is done, the 2FA code will be sent to the server together with the JWT token (first partial authentication) obtained in the previous step. The server checks the data. If these are invalid, the server responds with the corresponding error code. If the data is correct, the server responds with the user's index 0 public key + JWT token (second partial authentication – special case: different from the first – only used for the password lost functionality).

Interface to request the setting of a new password		
Method	**Description**	**Response**
lost_password_tfa **POST** /portal/user/ auth/lost_password_tfa	The user must enter the 2FA code in order to reset the password **Parameters:** • tfa_code ○ mandatory ○ current 2FA code of the user • JWT token (first partial authentication)	**LostPasswordTfa Response**
Data Objects		
Name	**JSON**	

LostPasswordTfa Response
On status 200
	{
	“public_key_0”: public key 0 for the user
	}

ValidationError
On Status 400 or 500
	{
	“error_code”: 1000,
	“parameter_name”: “tfa_code”,
	“error_message”: “invalid 2FA code”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. If the email has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.

Table 23 – server interface for confirming the 2fa code for password reset

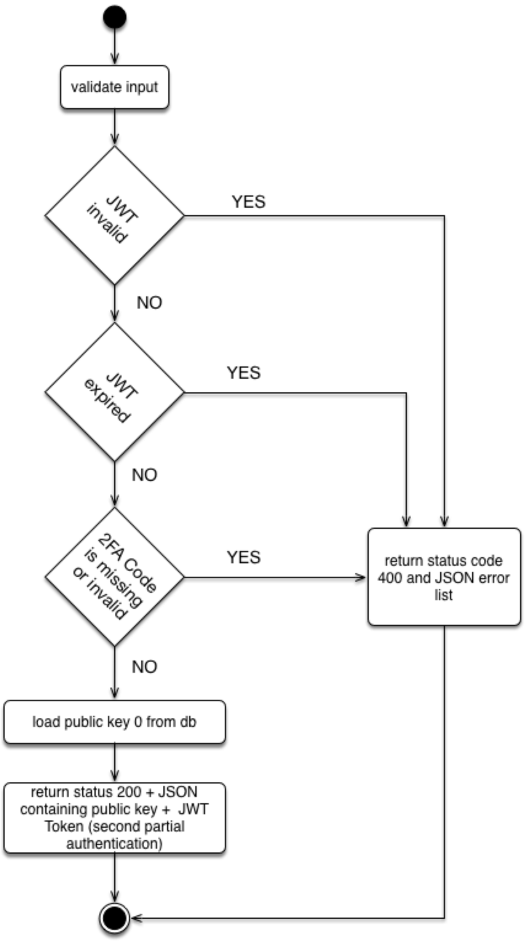

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
2FA Code is missing or invalid
YES
NO
return status code 400 and JSON error list
load public key 0 from db
return status 200 + JSON containing public key + JWT Token (second partial authentication)

Image 17 – Implementation overview: password lost step 3

Then the client asks the user to enter his mnemonic. The user enters his mnemonic and the client derives from the inserted mnemonic the public key of index 0. If the key matches the public key received from the server, then it is the correct mnemonic. Next the client shuffles a new wordlist and finds the indexes for the words of the mnemonic inserted by the user. If it cannot find one or more words it adds the words to the wordlist and shuffles it again. With the indexes its creates them mnemonic index list.

Next, the user can enter his new password. The client checks the password for its complexity and if this is good enough, the client generates two new random master key with the corresponding master key IVs, as it does when registering (see chapter Preparing and encrypting the payment relevant data), encrypts the mnemonic (list of indexes) and new shuffled word list with the corresponding master key, generates the new KDF password with new KDF salt and master key encryption IVs and encrypts the master keys with the KDF password. The KDF password salt, the encrypted mnemonic master key, the mnemonic master key encryption IV, the encrypted mnemonic and mnemonic encryption IV, the encrypted wordlist master key, the wordlist master key IV and the encrypted wordlist are then sent to the server together with the public key of 188 and the JWT token (second partial authentication) obtained in the previous steps.

Use this API endpoint to update the security data:

Interface to request the setting of a new password		
Method	**Description**	**Response**
lost_password_update **POST** (/portal/user/auth2 /lost_password_update	The call updates the security data for the user **Parameters:** • kdf_salt ○ mandatory • mnemonic_master_key ○ mandatory • mnemonic_master_iv ○ mandatory • encrypted_mnemonic ○ mandatory • encryption_mnemonic_iv ○ mandatory • wordlist_master_key ○ mandatory • wordlist_master_iv ○ mandatory • encrypted_wordlist ○ mandatory • encryption_wordlist_iv ○ mandatory • public_key_0 ○ mandatory • public_key_188 ○ mandatory	**LostPasswordUpdate Response**
Data Objects		
Name	**JSON**	

LostPasswordUpdate Response
On status 200
	{
	}	

ValidationError **On Status 400 or 500**	{ “error_code”: 1002, “parameter_name”: “kdf_salt”,	

	“error_message”: “missing parameter kdf salt” }
Possible errors	
Error code	**Description**
1000	Invalid argument. If the email has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.

Table 24 – server interface for updating the user security data (from password change)

The server checks the validity of the data. He first checks the JWT token, then the public key at index 188 and then the validity of the other data (KDF salt, the encrypted master keys, the master key encryption IVs, the encrypted mnemonic and mnemonic encryption IV, the encrypted wordlist, the wordlist encryption iv) like the validation in the registration process. If everything fits, he overwrites the data (KDF salt, the encrypted master keys, the master key encryption IVs, the encrypted mnemonic, mnemonic encryption IV, the encrypted wordlist, and the wordlist encryption IV) in the database and returns status 200 to the client. No full authentication, next the client shows the login form to the user.

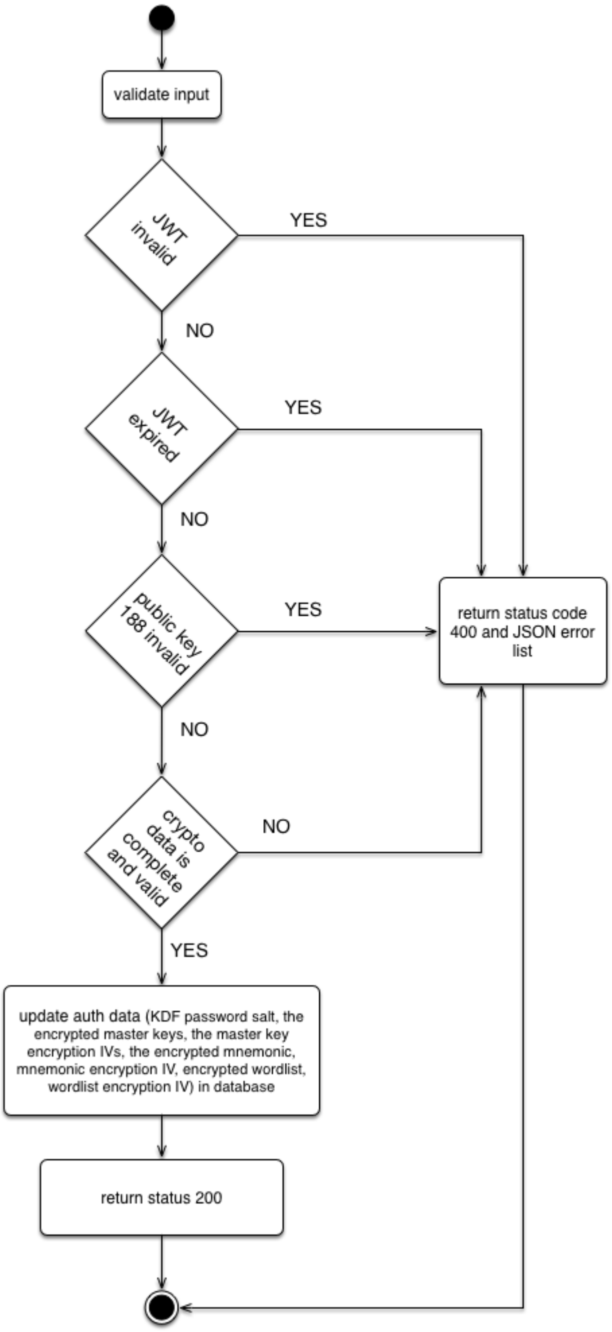

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
public key 188 invalid
YES
return status code 400 and JSON error list
NO
crypto data is complete and valid
NO
YES
update auth data (KDF password salt, the encrypted master keys, the master key encryption IVs, the encrypted mnemonic, mnemonic encryption IV, encrypted wordlist, wordlist encryption IV) in database
return status 200

Image 18 – Implementation overview: password lost step 4

The client can now confirm to the user that that password has been changed and can redirect him to the login form.

If the user has an unconfirmed mnemonic or unconfirmed 2fa registration the client needs to generate a new mnemonic and related data for the user (like in the registration process). In this case the user must not enter his mnemonic but just a new password. The client encrypts the new data and sends it to the server for update. If the mnemonic is unconfirmed but the 2fa registration is confirmed, the client must ask the user for the 2fa code first, before requesting the new password.

@Udo: please add interface description for the update described above

4.11 Change password

The user can change his password only if he is logged in. To change the password, he selects the corresponding menu item in the client.

Next, the client uses the JWT token (fully authenticated) received from the server on login to request the authentication data (KDF salt, encrypted master keys, master key encryption IVs, the encrypted mnemonic, mnemonic encryption IV, encrypted wordlist, wordlist encryption IV and the public key of index 0) from the server, using this API endpoint:

Interface to request the user’s authentication data		
Method	**Description**	**Response**
GET (/portal/user/dashboard/user_auth_data	The call returns the security data for the logged in user **Parameters:** ○ JWT token (fully authenticated)	**UserSecurityData Response**
Data Objects		
Name	**JSON**	

UserAuthDataResponse
On status 200
	{
	“kdf_password_salt”: string,
	“encrypted_mnemonic_master_key”: string,
	“mnemonic_master_key_encryption_iv”: string,
	“encrypted_mnemonic”: string,
	“mnemonic_encryption_iv”: string,
	“encrypted_wordlist_master_key”: string,
	“wordlist_master_key_encryption_iv” : string,
	“encrypted_wordlist” : string,
	“wordlist_encryption_iv”: string
	“public_key_index0”: string
	}	

ValidationError
On Status 400 or 500
	{
	}	

Possible errors		
Error code	**Description**	

Table 25 – server interface for retrieving the user security data

If the JWT token is correct, the server sends the data to the client.

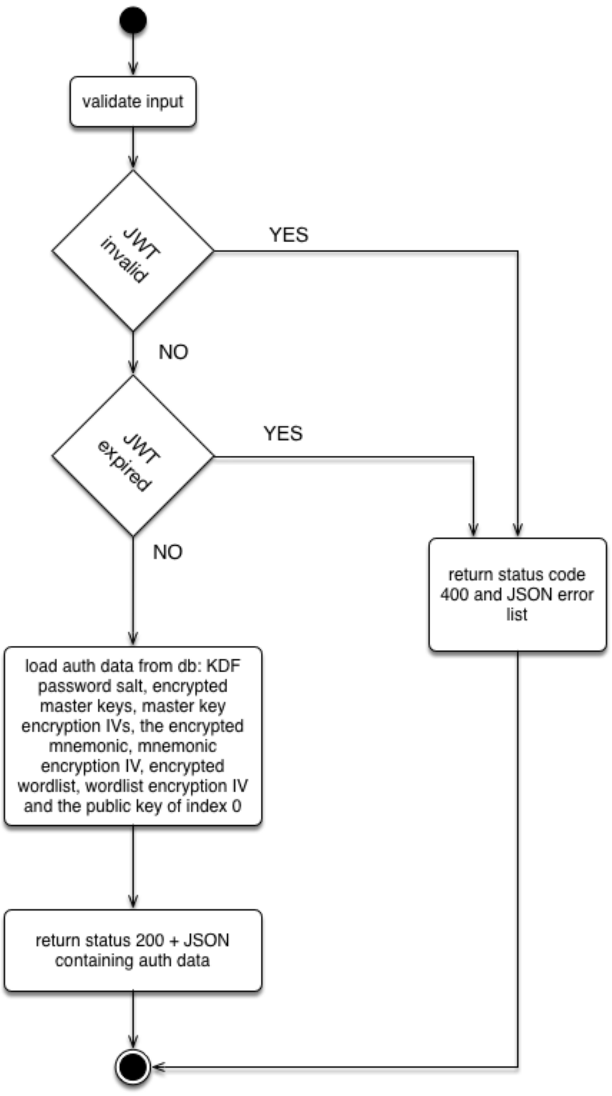

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
return status code 400 and JSON error list
load auth data from db: KDF password salt, encrypted master keys, master key encryption IVs, the encrypted mnemonic, mnemonic encryption IV, encrypted wordlist, wordlist encryption IV and the public key of index 0
return status 200 + JSON containing auth data

Image 19 – Implementation overview: change password step 1

Next, the client asks the user for the current and for the old, the new and the repetition of the new password. Once the user has entered the two passwords, the client first validates if the new password and its repetition are the same. Then it checks the current password entered by the user. If the current password is correct, the client generates a new KDF password salt and uses the new password to derive a new KDF password with the new salt. With the new KDF password, it generates a new master key encryption IV and encrypts the master keys with the new KDF password and master key encryption IVs. Then it sends the newly generated KDF password salt, the generated master key encryption IVs, the newly encrypted master keys, with the JWT token and the public key of index 188 (to prove that the inserted old password was correct) to the server. The mnemonic and wordlist are not re-encrypted, it is enough to re-encrypt the master keys. In addition, the client deletes the user's passwords, the KDF passwords, the public key of index 188, the decrypted and encrypted master keys and the decrypted and encrypted mnemonic and wordlist from its memory and does not store any of them in his internal storage/database. To send the data to the server, the client uses following interface:

Interface to request the changing of the password		
Method	**Description**	**Response**
change_password **POST** (/portal/user/dashboard/ change_password	The call changes the security data for the user, based on the new password he provided on the frontend. **Parameters:** • kdf_salt ○ mandatory • mnemonic_master_key ○ mandatory • mnemonic_master_iv ○ mandatory • wordlist_master_key ○ mandatory • wordlist_encryption_iv ○ mandatory • public_key_188 ○ mandatory • JWT token (fully authenticated)	**ChangePassword Response**
Data Objects		
Name	**JSON**	

ChangePasswordResponse
On status 200
	{
	}	

ValidationError
On Status 400 or 500
	{
	    “error_code”: 1013,
	    “parameter_name”: “public_key_188”,
	    “error_message”: “Public key 188 does not match”
	}	

Possible errors		
Error code	**Description**	
1013	Returned, if the public key 188 does not match to the one saved in the backend.	

Table 26 – server interface for updating the user security data

The server checks the JWT token and public key at index 188 and if it is correct it updates the master keys and IVs in the database. Then, the server tells the success to the client and the client informs the user that his password has been changed.

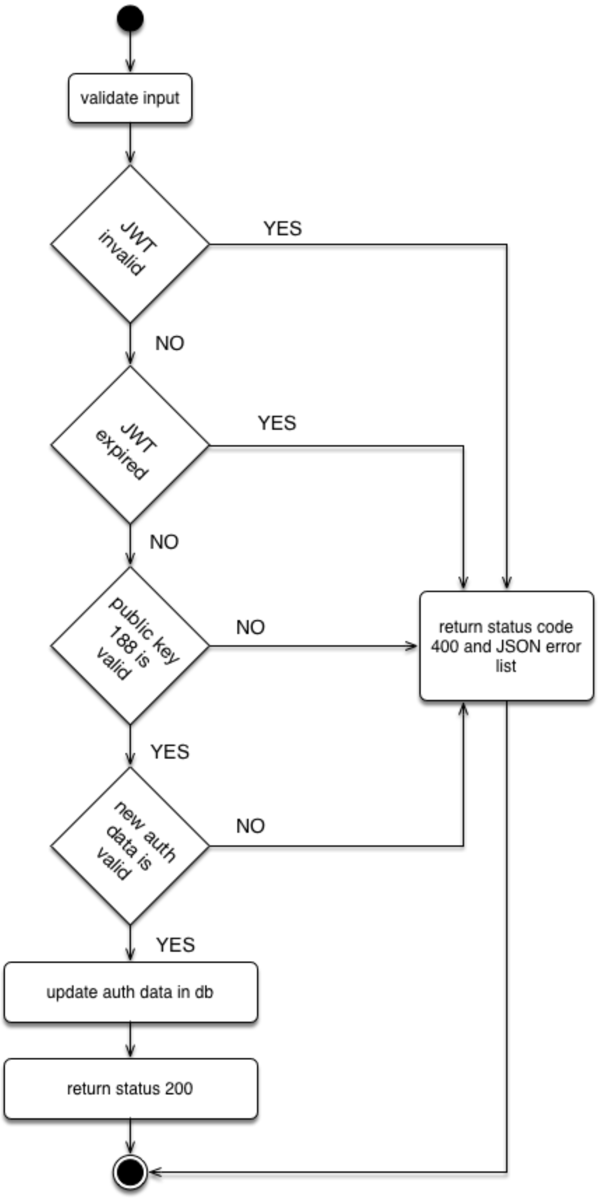


Image 20 – Implementation overview: change password step 2

The client can now let the user know that the password has been successfully changed.

4.12 2FA secret lost

As already described in the overview, the user can request a new 2FA secret if he has lost his 2FA secret. To do this, he presses the button "2FA secret lost". The client asks the user to enter his email address first. After the user has entered his email address, the client sends it to the server. He uses the following interface:

Interface to request the setting of a new password		
Method	**Description**	**Response**
lost_tfa **POST** /portal/user/lost_tfa	The call initiates the reset 2FA functionality **Parameters:** • email ○ mandatory	**LostTfa Response**

Data Objects	
Name	**JSON**

LostTfa Response
On status 200
	{
	}

ValidationError
On Status 400 or 500
	{
	“error_code”: 1010,
	“parameter_name”: “email”,
	“error_message”: “Email address is not confirmed”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. If the email has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.
1010	Returned, if the email was not yet confirmed

Table 27 – server interface for initiating the 2FA lost functionality

The server checks the email address and if it exists and is confirmed, it sends an email to the user with the link to reset the 2FA secret and returns status 200 to the client. If the email address does not exist or is not confirmed, the server returns the corresponding error to the client.

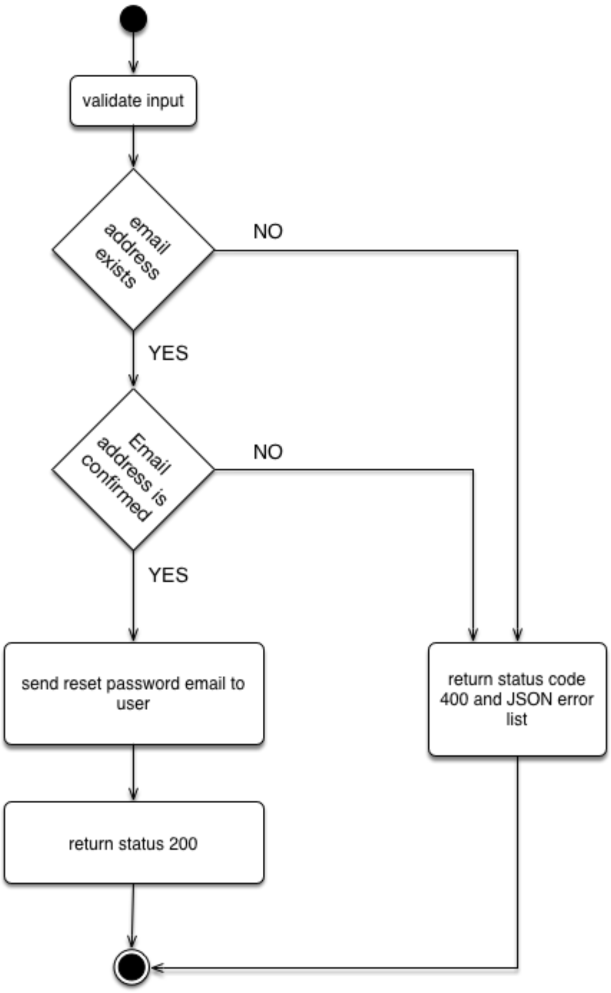

validate input
email address exists
NO
YES
Email address is confirmed
NO
YES
send reset password email to user
return status code 400 and JSON error list
return status 200

Image 21 – Implementation overview: 2FA secret lost step 1

If the server sent the reset 2FA secret email and the user presses the reset 2FA secret link in the received mail, a new web-client forwards the token from the link to the server. This API endpoint is described in the chapter Confirm token by mail.

The server checks the token, thus ensuring that the user has access to the specified email address. If the token is invalid or expired, the server returns the corresponding error code to the client. If the token is correct, the server partially logs in the user, loads the data required by the client for password validation (KDF password salt, the encrypted mnemonic master key, the mnemonic master key encryption IV, the encrypted mnemonic and mnemonic encryption IV, the encrypted wordlist master key, the wordlist master key IV, the encrypted wordlist and the wordlist encryption IV + public key of index 0) and returns it together with the JWT token (partially authenticated) to the client.

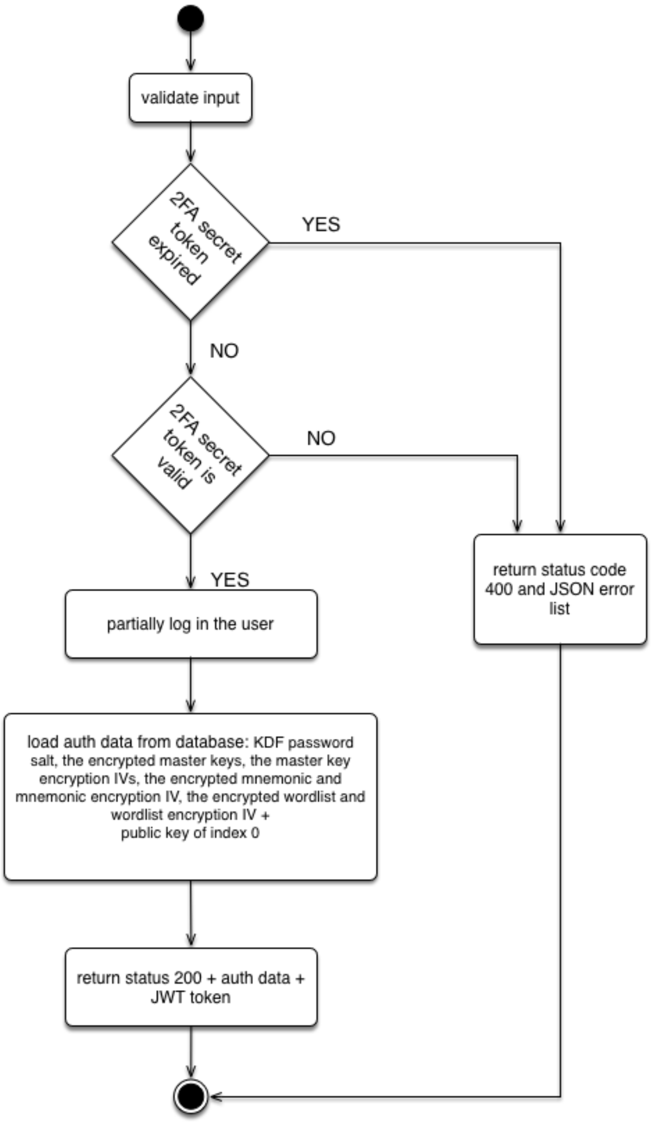

validate input
2FA secret token expired
YES
NO
2FA secret token is valid
NO
YES
partially log in the user
return status code 400 and JSON error list
load auth data from database: KDF password salt, the encrypted master keys, the master key encryption IVs, the encrypted mnemonic and mnemonic encryption IV, the encrypted wordlist and wordlist encryption IV + public key of index 0
return status 200 + auth data + JWT token

Image 22 – Implementation overview: 2FA secret lost step 2

Next, the client requires the user to enter his password. Then the client checks the password of the user. With the help of the password the client decrypts the data he has received from the server, generates from the decrypted mnemonic the public key of index 0 and compares it with the public key of index 0 obtained from the server. If these match, the client can assume that the entered password is correct. To prove this to the server, it generates the public key of index 188 and sends it together with the previously received JWT token (partially authenticated) to the server to fully authenticate the user and to request a new 2FA secret.

Interface to get a new 2FA secret		
Method	**Description**	**Response**
new_2fa_secret **POST** /portal/user/auth/ new_2fa_secret	The call initiates the reset 2FA functionality **Parameters:** • public_key_188 ○ mandatory ○ Publickey 188 of the user	**NewTfaSecretResponse**

	- JWT token (partially authenticated)	
Data Objects		
Name	**JSON**	

NewTfaSecretResponse
On status 200
	{
	“tfa_secret”: string, /* new 2FA secrete */
	“tfa_qr_image”: string, base64 encoded byte data /* new 2FA qr image */
	}	

ValidationError
On Status 400 or 500
	{
	“error_code”: 1013,
	“parameter_name”: “public_key_188”,
	“error_message”: “Password does not match”
	}	

Possible errors		
Error code	**Description**	
1002	Missing mandatory parameter.	
1013	Password does not match	

Table 28 – server interface for recreating new 2FA data

The server checks the JWT token (partially authenticated) and the obtained public key of index 188. If these are not okay, he returns the corresponding error to the client. If the data is correct, the server generates a new 2FA secret and overwrites the current 2FA secret from the database with the new generated 2FA secret. It also sets the 2fa confirmed flag to false in the database. Then the server generates the QR image and passes it to the client together with the new 2FA secret.

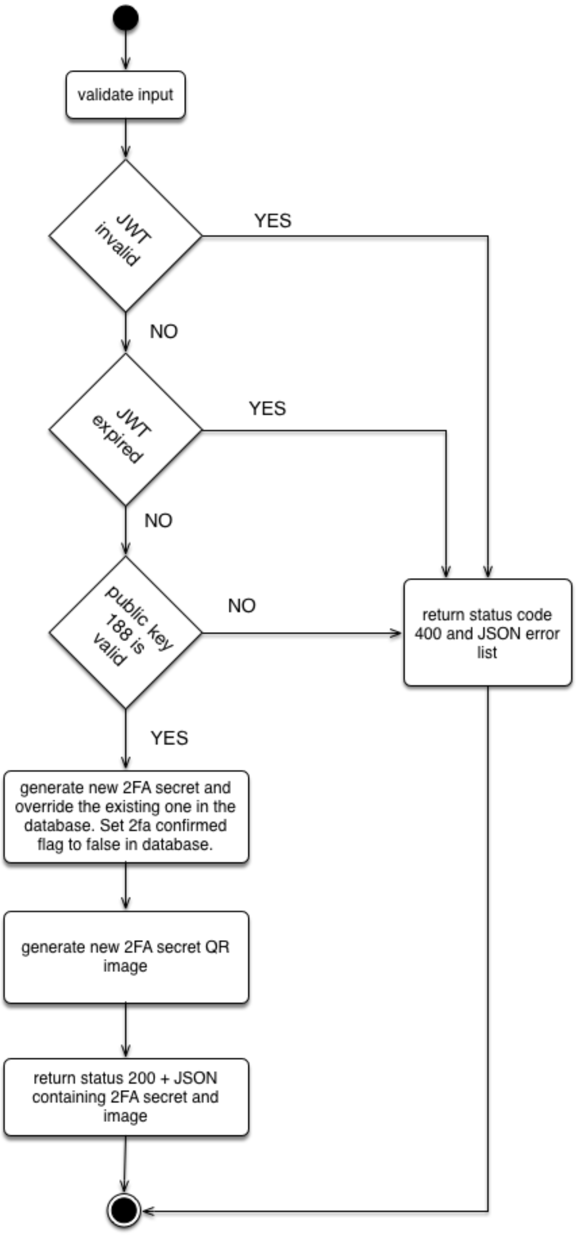

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
public key 188 is valid
NO
YES
return status code 400 and JSON error list
generate new 2FA secret and override the existing one in the database. Set 2fa confirmed flag to false in database.
generate new 2FA secret QR image
return status 200 + JSON containing 2FA secret and image

Image 23 – Implementation overview: 2FA secret lost step 3

Next, the client displays the image and the secret and asks the user to use this and to enter the current 2FA code. The user enters the current 2FA code and the client sends it together with the JWT token (partially authenticated) to the server. To do so it uses the interface defined in chapter Performing the 2FA registration.

If the server accepts the input, the client can show the user the successful reset of the 2FA secret, delete his data and forward him to login.

4.13 Change 2FA Secret

To change his 2FA secret the user must be logged in. To change his 2FA secret, he presses the corresponding menu entry in the dashboard. First, the client requests the user's encrypted authentication data from the server with the help of the JWT token (fully authenticated) received at login. He uses the interface “user_auth_data” specified in the chapter Change password. Once the client has received the data, it asks the user to enter his password. Next, the client decrypts the authentication data with the password entered by the user and derives the public key from index 0 from the decrypted mnemonic. The key is then compared to the public key of index 0 received from the server, and if they match, the client can assume that the user has entered the correct password.

Next, the client requests a new 2FA secret from the server. It passes the JWT token (fully authenticated) from the login and the derived public key from index 188 to the server to prove to the server that the user has entered the correct password. To do this, it uses the server endpoint “new_2fa_secret” which is described in chapter 2FA secret lost.

The server checks the JWT token (fully authenticated) and the obtained public key of index 188. If these are not okay, he returns the corresponding error to the client. If the data is correct, the server generates a new 2FA secret and overwrites the current 2FA secret from the database with the new generated 2FA secret. It also sets the 2fa secret confirm flag to false. Then the server generates the 2FA secret QR image and passes it to the client together with the new 2FA secret.

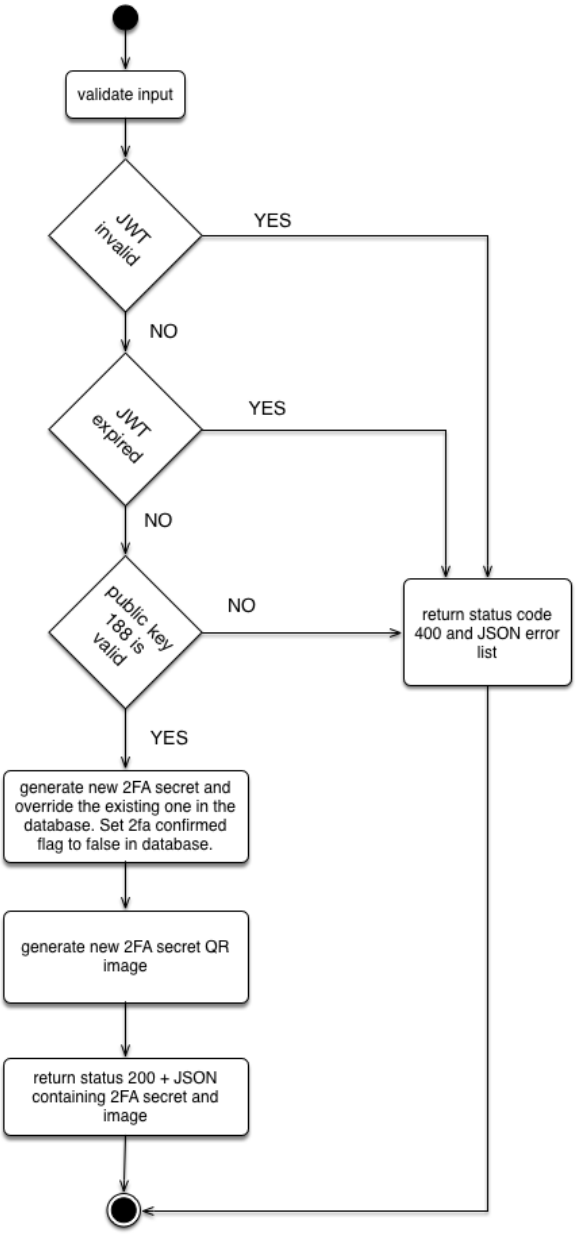

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
public key 188 is valid
NO
YES
return status code 400 and JSON error list
generate new 2FA secret and override the existing one in the database. Set 2fa confirmed flag to false in database.
generate new 2FA secret QR image
return status 200 + JSON containing 2FA secret and image

Image 24 – Implementation overview: 2FA secret lost step 3

Next, the client displays the image and the secret and asks the user to use this and to enter the current 2FA code. The user enters the current 2FA code and the client sends it together with the JWT token (fully authenticated) to the server.

For this, the server provides the following interface:

Interface to confirm 2FA registration		
Method	**Description**	**Response**
confirm_new_2fa_secret **POST** /portal/user/dashboard/confirm_new_2fa_secret	After scanning the QR-image or directly inserting the 2FA secret into an authenticator app e.g. “Google Authenticator”, the user confirms the new 2FA secret by inserting the current 2FA code. **Parameters:** • tfa_code	**ConfirmNewTfaSecretResponse**

	○ mandatory ○ the current 2FA code • JWT Token (fully authenticated) ○ mandatory ○ The JWT token that the client received on login.	

Data Objects	
Name	**JSON**

ConfirmNewTfaSecretResponse on status 200	{
	mail_confirmed: bool,
	tfa_confirmed: bool,
	mnemonic_confirmed: bool
	}

ValidationError
On status 400 or 500
	{
	“error_code”: 1000, /* mandatory, possible values: see table below */
	/* name of the parameter, see above */
	“parameter_name”: “tfa_code”,
	/* message for the client/user, if needed */
	“user_error_message_key”: “confirm_2FAregsitration.2FACode.invalid”
	/* optional, message for the developer from server*/
	“error_message”: “2FA code is invalid.”
	}

Possible errors	
Error code	**Description**
1000	Invalid argument. If the email or JWT token has a wrong format or could not be found in the database.
1002	Missing mandatory parameter.

Table 29 – server interface for confirming the new 2FA secret

Next, the server validates the received JWT token and the passed 2FA code. If one of the two is not correct, he returns the corresponding error. If both are correct, the server sets the 2fa confirmed flag in the database to true and returns status 200 to the client.

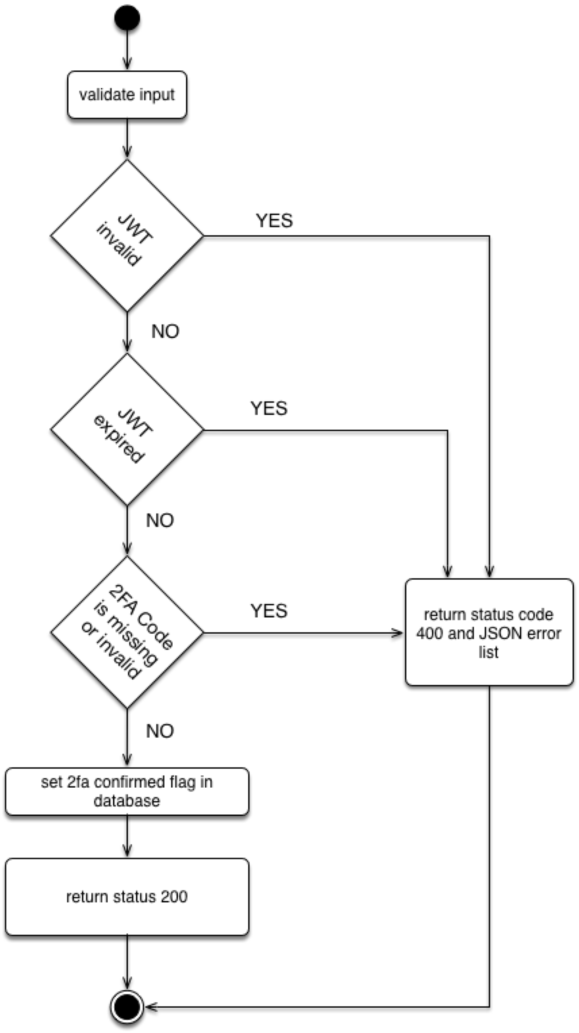

validate input
JWT invalid
YES
NO
JWT expired
YES
NO
2FA Code is missing or invalid
YES
NO
return status code 400 and JSON error list
set 2fa confirmed flag in database
return status 200

Image 25 – Implementation overview: reset 2fa secret - confirmation

The client can show the user the successful changing of the 2FA secret.

5. Error codes and keys

Possible error codes:

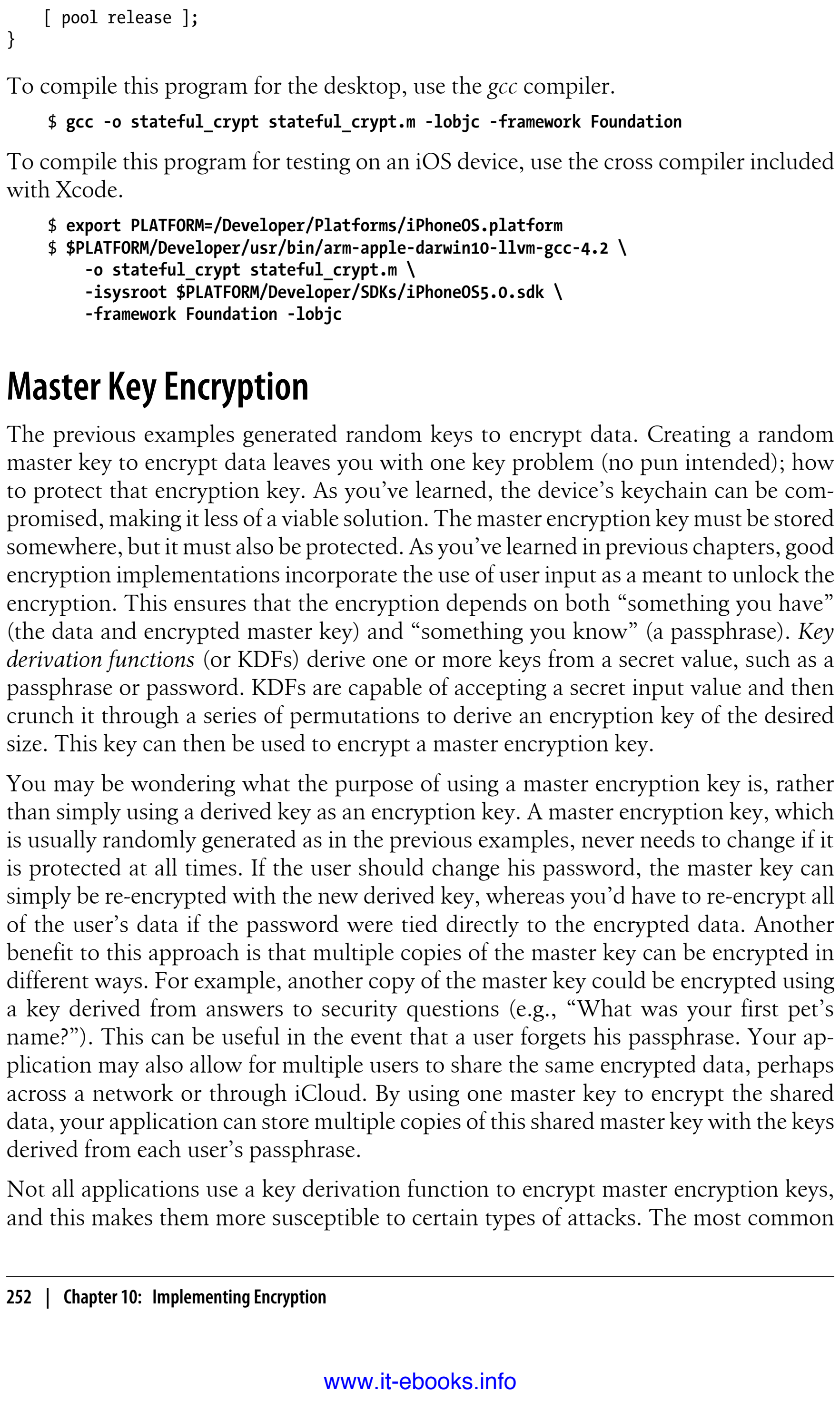
Error code	Description
1	General error
2	Bind-Error. This error is returned, if the data sent to an endpoint could not be bound to the data structures. Most likely the format of any data is wrong. If this error is returned, no details about the fields are included. Message will be "Bad input data"
1000	Invalid argument from a user input. If there is something wrong with one of the values given by the user.
1001	Email-address already exists in the database. An account could not be created.
1002	Missing mandatory parameter.

1003	Mobile number does not match country of residence.
1004	Invalid length of parameter value.
1005	Master key encryption IV is the same as mnemonic encryption IV.
1006	Used in the Registration process: Email-confirmation token expired.
1007	JWT Token expired
1008	Email already confirmed
1009	2FA already confirmed
1010	Email address not confirmed
1011	Mnemonic not confirmed
1012	2FA not yet confirmed
1013	Invalid password (public key of index 188)

Error message keys:

6. Appendix

a. Master Key Encryption



```
    [ pool release ];
}
```

To compile this program for the desktop, use the *gcc* compiler.

```
    $ gcc -o stateful_crypt stateful_crypt.m -lobjc -framework Foundation
```

To compile this program for testing on an iOS device, use the cross compiler included with Xcode.

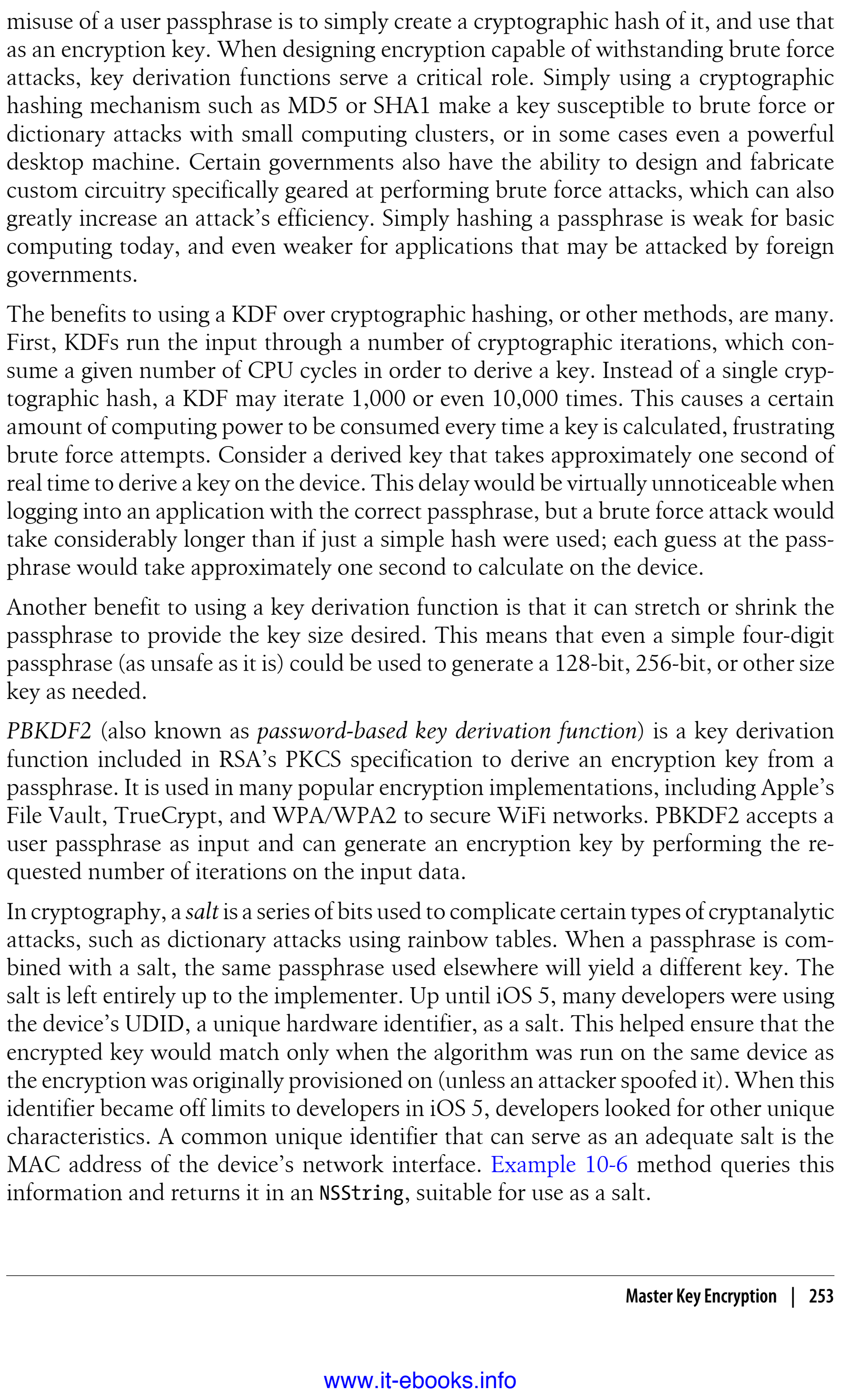
```
    $ export PLATFORM=/Developer/Platforms/iPhoneOS.platform
    $ $PLATFORM/Developer/usr/bin/arm-apple-darwin10-llvm-gcc-4.2 \
        -o stateful_crypt stateful_crypt.m \
        -isysroot $PLATFORM/Developer/SDKs/iPhoneOS5.0.sdk \
        -framework Foundation -lobjc
```

Master Key Encryption

The previous examples generated random keys to encrypt data. Creating a random master key to encrypt data leaves you with one key problem (no pun intended); how to protect that encryption key. As you've learned, the device's keychain can be compromised, making it less of a viable solution. The master encryption key must be stored somewhere, but it must also be protected. As you've learned in previous chapters, good encryption implementations incorporate the use of user input as a meant to unlock the encryption. This ensures that the encryption depends on both "something you have" (the data and encrypted master key) and "something you know" (a passphrase). *Key derivation functions* (or KDFs) derive one or more keys from a secret value, such as a passphrase or password. KDFs are capable of accepting a secret input value and then crunch it through a series of permutations to derive an encryption key of the desired size. This key can then be used to encrypt a master encryption key.

You may be wondering what the purpose of using a master encryption key is, rather than simply using a derived key as an encryption key. A master encryption key, which is usually randomly generated as in the previous examples, never needs to change if it is protected at all times. If the user should change his password, the master key can simply be re-encrypted with the new derived key, whereas you'd have to re-encrypt all of the user's data if the password were tied directly to the encrypted data. Another benefit to this approach is that multiple copies of the master key can be encrypted in different ways. For example, another copy of the master key could be encrypted using a key derived from answers to security questions (e.g., "What was your first pet's name?"). This can be useful in the event that a user forgets his passphrase. Your application may also allow for multiple users to share the same encrypted data, perhaps across a network or through iCloud. By using one master key to encrypt the shared data, your application can store multiple copies of this shared master key with the keys derived from each user's passphrase.

Not all applications use a key derivation function to encrypt master encryption keys, and this makes them more susceptible to certain types of attacks. The most common



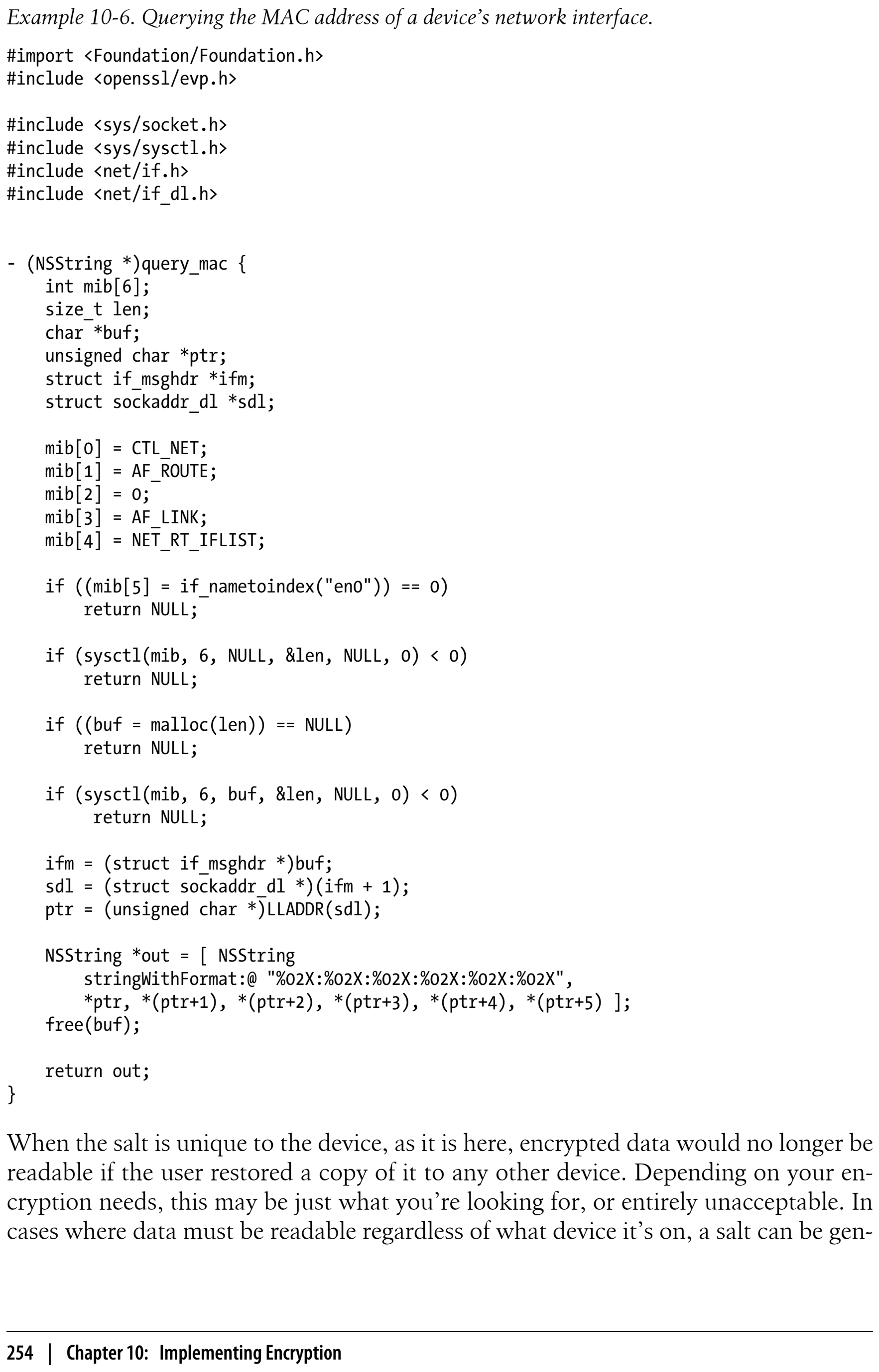
misuse of a user passphrase is to simply create a cryptographic hash of it, and use that as an encryption key. When designing encryption capable of withstanding brute force attacks, key derivation functions serve a critical role. Simply using a cryptographic hashing mechanism such as MD5 or SHA1 make a key susceptible to brute force or dictionary attacks with small computing clusters, or in some cases even a powerful desktop machine. Certain governments also have the ability to design and fabricate custom circuitry specifically geared at performing brute force attacks, which can also greatly increase an attack's efficiency. Simply hashing a passphrase is weak for basic computing today, and even weaker for applications that may be attacked by foreign governments.

The benefits to using a KDF over cryptographic hashing, or other methods, are many. First, KDFs run the input through a number of cryptographic iterations, which consume a given number of CPU cycles in order to derive a key. Instead of a single cryptographic hash, a KDF may iterate 1,000 or even 10,000 times. This causes a certain amount of computing power to be consumed every time a key is calculated, frustrating brute force attempts. Consider a derived key that takes approximately one second of real time to derive a key on the device. This delay would be virtually unnoticeable when logging into an application with the correct passphrase, but a brute force attack would take considerably longer than if just a simple hash were used; each guess at the passphrase would take approximately one second to calculate on the device.

Another benefit to using a key derivation function is that it can stretch or shrink the passphrase to provide the key size desired. This means that even a simple four-digit passphrase (as unsafe as it is) could be used to generate a 128-bit, 256-bit, or other size key as needed.

PBKDF2 (also known as *password-based key derivation function*) is a key derivation function included in RSA's PKCS specification to derive an encryption key from a passphrase. It is used in many popular encryption implementations, including Apple's File Vault, TrueCrypt, and WPA/WPA2 to secure WiFi networks. PBKDF2 accepts a user passphrase as input and can generate an encryption key by performing the requested number of iterations on the input data.

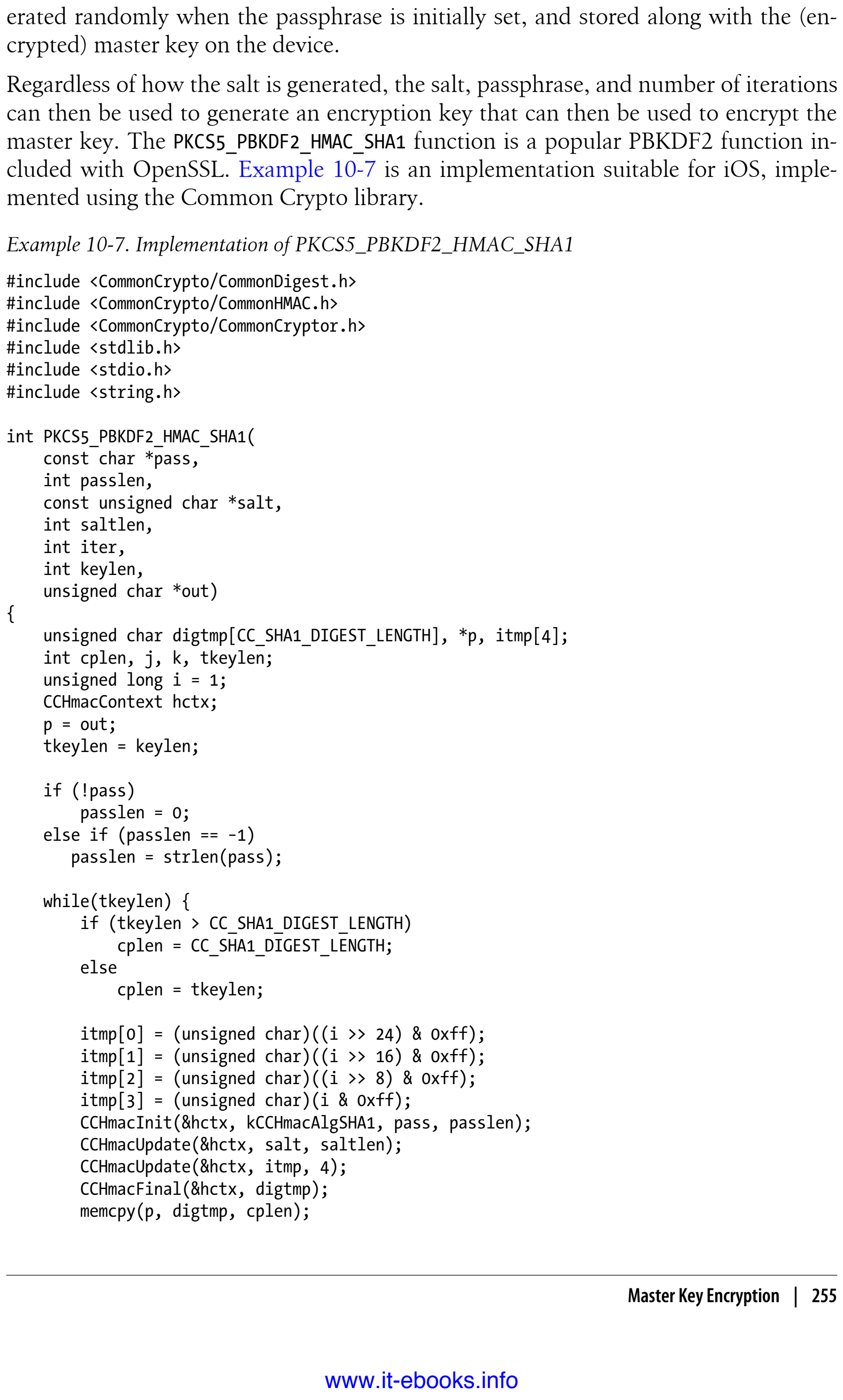
In cryptography, a *salt* is a series of bits used to complicate certain types of cryptanalytic attacks, such as dictionary attacks using rainbow tables. When a passphrase is combined with a salt, the same passphrase used elsewhere will yield a different key. The salt is left entirely up to the implementer. Up until iOS 5, many developers were using the device's UDID, a unique hardware identifier, as a salt. This helped ensure that the encrypted key would match only when the algorithm was run on the same device as the encryption was originally provisioned on (unless an attacker spoofed it). When this identifier became off limits to developers in iOS 5, developers looked for other unique characteristics. A common unique identifier that can serve as an adequate salt is the MAC address of the device's network interface. Example 10-6 method queries this information and returns it in an `NSString`, suitable for use as a salt.



Example 10-6. Querying the MAC address of a device's network interface.

```
#import <Foundation/Foundation.h>
#include <openssl/evp.h>

#include <sys/socket.h>
#include <sys/sysctl.h>
#include <net/if.h>
#include <net/if_dl.h>


- (NSString *)query_mac {
    int mib[6];
    size_t len;
    char *buf;
    unsigned char *ptr;
    struct if_msghdr *ifm;
    struct sockaddr_dl *sdl;

    mib[0] = CTL_NET;
    mib[1] = AF_ROUTE;
    mib[2] = 0;
    mib[3] = AF_LINK;
    mib[4] = NET_RT_IFLIST;

    if ((mib[5] = if_nametoindex("en0")) == 0)
        return NULL;

    if (sysctl(mib, 6, NULL, &len, NULL, 0) < 0)
        return NULL;

    if ((buf = malloc(len)) == NULL)
        return NULL;

    if (sysctl(mib, 6, buf, &len, NULL, 0) < 0)
         return NULL;

    ifm = (struct if_msghdr *)buf;
    sdl = (struct sockaddr_dl *)(ifm + 1);
    ptr = (unsigned char *)LLADDR(sdl);

    NSString *out = [ NSString
        stringWithFormat:@ "%02X:%02X:%02X:%02X:%02X:%02X",
        *ptr, *(ptr+1), *(ptr+2), *(ptr+3), *(ptr+4), *(ptr+5) ];
    free(buf);

    return out;
}
```

When the salt is unique to the device, as it is here, encrypted data would no longer be readable if the user restored a copy of it to any other device. Depending on your encryption needs, this may be just what you're looking for, or entirely unacceptable. In cases where data must be readable regardless of what device it's on, a salt can be gen-

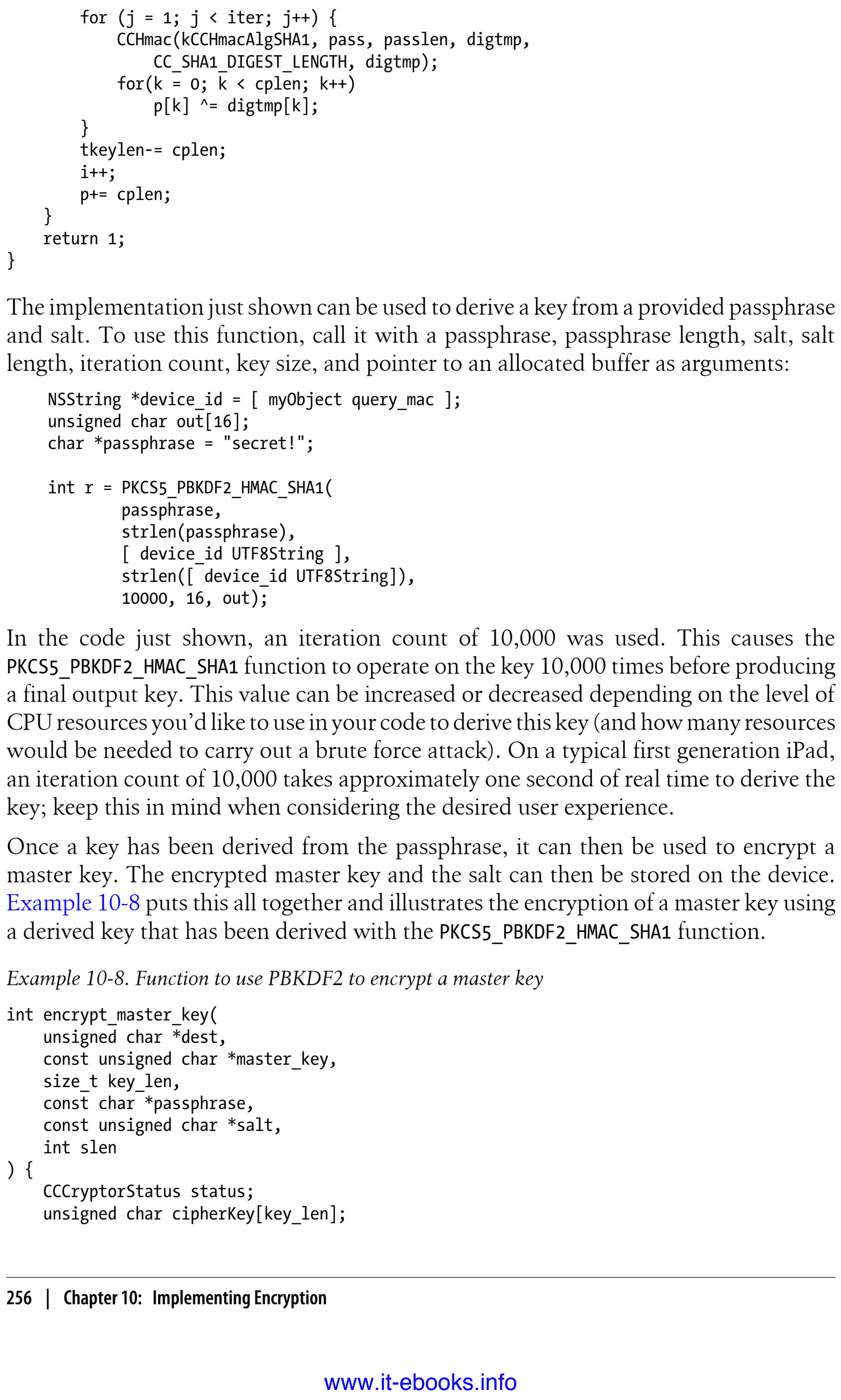


erated randomly when the passphrase is initially set, and stored along with the (encrypted) master key on the device.

Regardless of how the salt is generated, the salt, passphrase, and number of iterations can then be used to generate an encryption key that can then be used to encrypt the master key. The `PKCS5_PBKDF2_HMAC_SHA1` function is a popular PBKDF2 function included with OpenSSL. Example 10-7 is an implementation suitable for iOS, implemented using the Common Crypto library.

Example 10-7. Implementation of PKCS5_PBKDF2_HMAC_SHA1

```
#include <CommonCrypto/CommonDigest.h>
#include <CommonCrypto/CommonHMAC.h>
#include <CommonCrypto/CommonCryptor.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int PKCS5_PBKDF2_HMAC_SHA1(
    const char *pass,
    int passlen,
    const unsigned char *salt,
    int saltlen,
    int iter,
    int keylen,
    unsigned char *out)
{
    unsigned char digtmp[CC_SHA1_DIGEST_LENGTH], *p, itmp[4];
    int cplen, j, k, tkeylen;
    unsigned long i = 1;
    CCHmacContext hctx;
    p = out;
    tkeylen = keylen;

    if (!pass)
        passlen = 0;
    else if (passlen == -1)
       passlen = strlen(pass);

    while(tkeylen) {
        if (tkeylen > CC_SHA1_DIGEST_LENGTH)
            cplen = CC_SHA1_DIGEST_LENGTH;
        else
            cplen = tkeylen;

        itmp[0] = (unsigned char)((i >> 24) & 0xff);
        itmp[1] = (unsigned char)((i >> 16) & 0xff);
        itmp[2] = (unsigned char)((i >> 8) & 0xff);
        itmp[3] = (unsigned char)(i & 0xff);
        CCHmacInit(&hctx, kCCHmacAlgSHA1, pass, passlen);
        CCHmacUpdate(&hctx, salt, saltlen);
        CCHmacUpdate(&hctx, itmp, 4);
        CCHmacFinal(&hctx, digtmp);
        memcpy(p, digtmp, cplen);
```



```
        for (j = 1; j < iter; j++) {
            CCHmac(kCCHmacAlgSHA1, pass, passlen, digtmp,
                CC_SHA1_DIGEST_LENGTH, digtmp);
            for(k = 0; k < cplen; k++)
                p[k] ^= digtmp[k];
        }
        tkeylen-= cplen;
        i++;
        p+= cplen;
    }
    return 1;
}
```

The implementation just shown can be used to derive a key from a provided passphrase and salt. To use this function, call it with a passphrase, passphrase length, salt, salt length, iteration count, key size, and pointer to an allocated buffer as arguments:

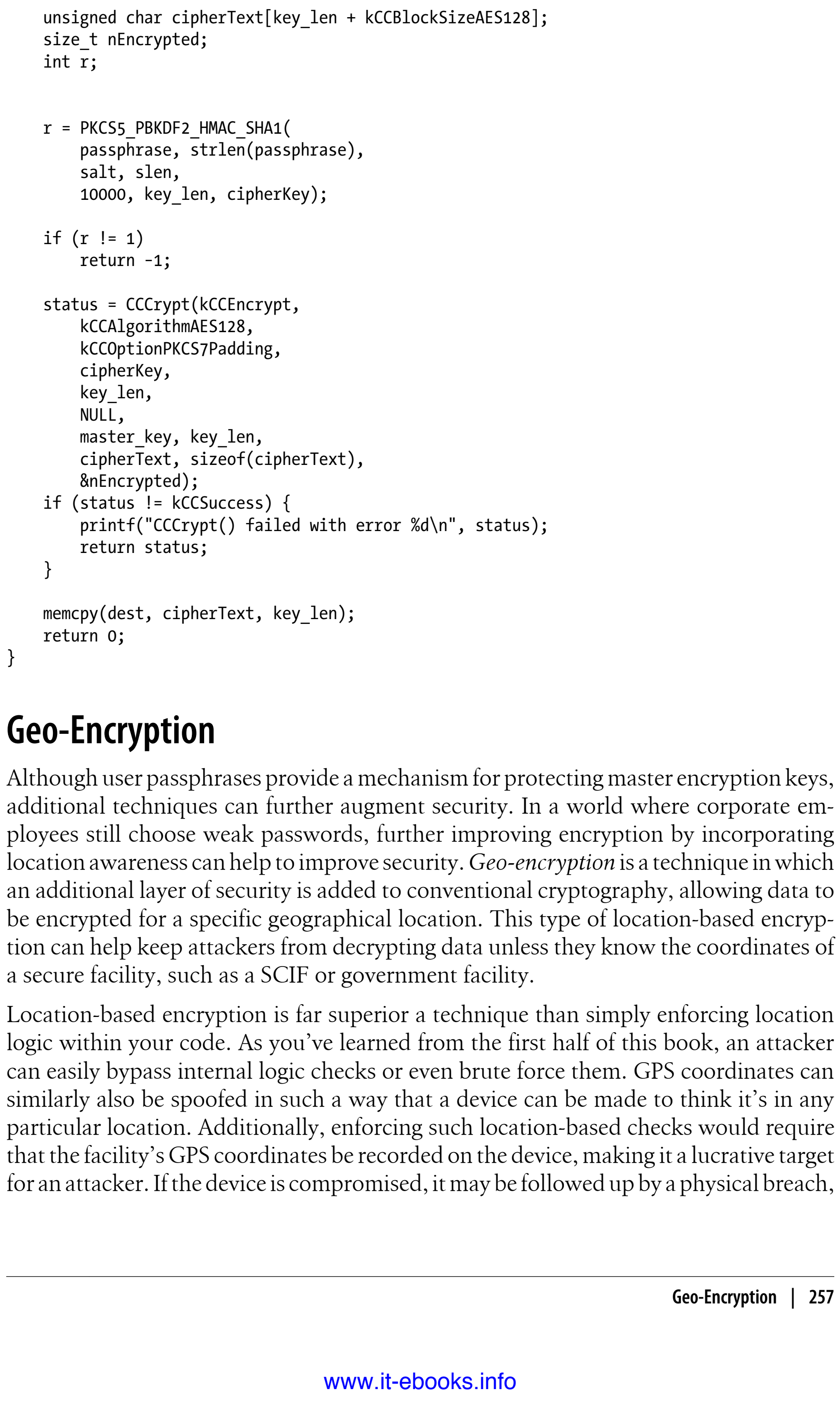
```
    NSString *device_id = [ myObject query_mac ];
    unsigned char out[16];
    char *passphrase = "secret!";

    int r = PKCS5_PBKDF2_HMAC_SHA1(
            passphrase,
            strlen(passphrase),
            [ device_id UTF8String ],
            strlen([ device_id UTF8String]),
            10000, 16, out);
```

In the code just shown, an iteration count of 10,000 was used. This causes the `PKCS5_PBKDF2_HMAC_SHA1` function to operate on the key 10,000 times before producing a final output key. This value can be increased or decreased depending on the level of CPU resources you'd like to use in your code to derive this key (and how many resources would be needed to carry out a brute force attack). On a typical first generation iPad, an iteration count of 10,000 takes approximately one second of real time to derive the key; keep this in mind when considering the desired user experience.

Once a key has been derived from the passphrase, it can then be used to encrypt a master key. The encrypted master key and the salt can then be stored on the device. Example 10-8 puts this all together and illustrates the encryption of a master key using a derived key that has been derived with the `PKCS5_PBKDF2_HMAC_SHA1` function.

Example 10-8. Function to use PBKDF2 to encrypt a master key

```
int encrypt_master_key(
    unsigned char *dest,
    const unsigned char *master_key,
    size_t key_len,
    const char *passphrase,
    const unsigned char *salt,
    int slen
) {
    CCCryptorStatus status;
    unsigned char cipherKey[key_len];
```



```
    unsigned char cipherText[key_len + kCCBlockSizeAES128];
    size_t nEncrypted;
    int r;


    r = PKCS5_PBKDF2_HMAC_SHA1(
        passphrase, strlen(passphrase),
        salt, slen,
        10000, key_len, cipherKey);

    if (r != 1)
        return -1;

    status = CCCrypt(kCCEncrypt,
        kCCAlgorithmAES128,
        kCCOptionPKCS7Padding,
        cipherKey,
        key_len,
        NULL,
        master_key, key_len,
        cipherText, sizeof(cipherText),
        &nEncrypted);
    if (status != kCCSuccess) {
        printf("CCCrypt() failed with error %d\n", status);
        return status;
    }

    memcpy(dest, cipherText, key_len);
    return 0;
}
```

Geo-Encryption

Although user passphrases provide a mechanism for protecting master encryption keys, additional techniques can further augment security. In a world where corporate employees still choose weak passwords, further improving encryption by incorporating location awareness can help to improve security. *Geo-encryption* is a technique in which an additional layer of security is added to conventional cryptography, allowing data to be encrypted for a specific geographical location. This type of location-based encryption can help keep attackers from decrypting data unless they know the coordinates of a secure facility, such as a SCIF or government facility.

Location-based encryption is far superior a technique than simply enforcing location logic within your code. As you've learned from the first half of this book, an attacker can easily bypass internal logic checks or even brute force them. GPS coordinates can similarly also be spoofed in such a way that a device can be made to think it's in any particular location. Additionally, enforcing such location-based checks would require that the facility's GPS coordinates be recorded on the device, making it a lucrative target for an attacker. If the device is compromised, it may be followed up by a physical breach,

Geo-Encryption | 257

1. Tables

Table 1 – general security requirements.....18
Table 2 – structure of returned error data.....20
Table 3 – structure of returned data for the user message list.....21
Table 4 – field description for the user’s message list.....21
Table 5 – structure of returned data for the user message.....21
Table 6 – field description for the user message.....22
Table 7 – List of endpoints returning simple authenticated header.....23
Table 8 – List of endpoints returning full authenticated header.....23
Table 9 – List of endpoints returning lost password authenticated header.....23
Table 10 – possible registration data.....25
Table 11 – validation of registration data.....26
Table 12 – server interfaces needed to display registration form.....27
Table 13 – server interface for registration form.....32
Table 14 – server interface for resending confirmation mail.....36
Table 15 – server interface for confirming the 2FA registration.....38
Table 16 – server interface for the retrieving the user details.....41
Table 17 – server interface for the first login step.....43
Table 18 – server interface for the second login step.....45
Table 19 – server interface to request the 2fa secret.....48
Table 20 – server interface for confirm token.....50
Table 21 – server interface for confirming the writing down of the mnemonic.....51
Table 22 – server interface for password lost step 1.....53
Table 23 – server interface for confirming the 2fa code for password reset.....56
Table 24 – server interface for updating the user security data (from password change).....58
Table 25 – server interface for retrieving the user security data.....60
Table 26 – server interface for updating the user security data.....62
Table 27 – server interface for initiating the 2FA lost functionality.....64
Table 28 – server interface for recreating new 2FA data.....67
Table 29 – server interface for confirming the new 2FA secret.....71

2. Images

Image 1 –Rough overview of the registration process.....9
Image 2 –Overview of the login process.....11

Image 3 –Overview of password lost process 14
Image 4 –Overview of the change password process 15
Image 5 –Overview of the 2FA secret lost process 16
Image 6 – Implementation overview: server-interface user registration 34
Image 7 – Implementation overview: server-interface for email confirmation 35
Image 8 – Implementation overview: server-interface to resend email-confirmation 37
Image 9 – Implementation overview: server-interface to confirm 2FA registration 39
Image 10 – Implementation overview: registration flow in the client 41
Image 11 – Implementation overview: partially log in user with 2FA code 44
Image 12 – Implementation overview: completely log in user with 2FA code 46
Image 13 – Implementation overview: login flow in the client 49
Image 14 – Implementation overview: confirm mnemonic 52
Image 15 – Implementation overview: password lost step 1 54
Image 16 – Implementation overview: password lost step 2 55
Image 17 – Implementation overview: password lost step 3 56
Image 18 – Implementation overview: password lost step 4 59
Image 19 – Implementation overview: change password step 1 61
Image 20 – Implementation overview: change password step 2 63
Image 21 – Implementation overview: 2FA secret lost step 1 65
Image 22 – Implementation overview: 2FA secret lost step 2 66
Image 23 – Implementation overview: 2FA secret lost step 3 68
Image 24 – Implementation overview: 2FA secret lost step 3 70
Image 25 – Implementation overview: reset 2fa secret - confirmation 72