Design and Prototypical Implementation of an Open Source and Smart Contract-based Know Your Customer (KYC) Platform

Sebastian Emmanuel Allemann
Zurich, Switzerland
Student ID: 14-721-807

Bachelor Thesis
Communication Systems Group, Prof. Dr. Burkhard Stiller

Supervisor: Sina Rafati, Eder John Scheid
Date of Submission: January 31, 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Für Finanzinstitutionen ist es eine Notwendigkeit, ihre Kunden zu kennen. Know Your Customer (KYC) Prozesse wurden zu diesem Zweck entwickelt. Mit der Zunahme der Blockchain Technologie, wird es immer wichtiger die Identitäten von Personen bestätigen zu können, da man nur der Technologie, ohne kontrollierende Drittpartei, vertraut. Personen müssen bei jeder Bank, mit welcher sie eine Geschäftsbeziehung aufnehmen möchten, aufs Neue einen KYC Prozess durchlaufen. Dies ist wenig effizient und kostet viel Zeit und Geld. Es sollte daher möglich sein, eine Plattform aufzubauen, welche KYC Prozesse als Dienstleistung anbietet. Diese könnte durch Smart Contracts identifizierte Personen bestätigen und dadurch die Effizienz steigern.
Das Ziel dieser Arbeit ist es, ein solches System in Form eines Prototypen zu entwickeln. Der Video Identifikationsprozess ist von der Eidgenössischen Finanzmarktaufsicht FINMA reguliert und definiert die Anforderungen, die zu berücksichtigen sind. Das Resultat des Identifikationsprozesses liefert einen Beweis für die Identität in Form eines einzigartigen Schlüssels. Zusätzlich verarbeitet ein erfolgreich entwickelter Smart Contract alle Anfragen von externen Plattformen, die den soeben genannten Schlüssel verifizieren möchten, wenn ein Kunde sich mit dem Schlüssel registriert.

Abstract

For financial institutions it is a necessity to know their customers. Know Your Customer (KYC) processes have been developed to meet this requirement. With the growing trend of blockchain technology, the proof of identification becomes even more important, since there is no controlling third party. Individuals need to go through a KYC process at each bank they wish to establish a business relationship with. This is not very efficient because it is time consuming and expensive. It should be possible to build a platform that provides KYC as a service and processes these through Smart Contract requests to authenticate identified persons. This would save both cost and time.
The goal of this thesis is to develop such a system in the form of a prototype platform. The video identification process is regulated by the Swiss Financial Market Supervisory Authority (FINMA) and defines the requirements that have to be considered. The result of the identification process is a proof of identity in the form of a unique key. In addition, a smart contract is implemented successfully, which processes all requests from external platforms that need to verify the just mentioned key.

Acknowledgments

I would like to offer my special thanks to my supervisor, Sina Rafati, for assistance and useful advice during my thesis.

I would like to thank Prof. Dr. Burkhard Stiller and the Communication Systems Group at the Department of Informatics at the University of Zurich for making this thesis possible.

My special thanks are extended to Arik Gabay for our successful collaboration throughout the thesis.

Contents

Zusammenfassung **i**

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Description of Work 1

1.2 Blockchain and Smart Contracts 2

1.3 Know Your Customer (KYC) 2

1.4 WebRTC 2

1.5 Thesis Outline 3

2 Related Work **5**

2.1 IDnow 5

2.2 CB Financial Services AG (CBFS) 6

2.3 Shufti Pro 6

2.4 Tradle 6

2.5 KYC-Chain 7

2.6 KYCstart 7

2.7 Comparison 7

3 Design **9**

3.1 Security Requirements 9

3.2 Video Identification Requirements 9

3.2.1 Smart Contract 11

4 Implementation **13**

4.1 JavaScript libraries 13

4.1.1 Server 13

4.1.2 Client 14

4.1.3 APIs 15

4.1.4 Browser Extensions 15

4.2 Concept of protection 15

4.2.1 Protection of pages 15

4.2.2 Email verification 17

4.2.3 Password Reset 18

4.2.4 Password Strength 19

4.3 Login 19

4.4 Video identification process 20

4.4.1 Registration 20

4.4.2 Email Confirmation 22

4.4.3 Beneficial Owners 22

4.4.4 Terms & Conditions 23

4.4.5 Ether Payment 24

4.4.6 Video Identification 25

4.4.7 OTP Verification 29

4.4.8 Profile 32

4.5 KYC key validation 33

4.6 Admin 34

4.7 System Architecture 35

5 The Smart Contract **37**

5.1 Functionality . . . 38

5.1.1 KYC Platform Address . . . 38

5.1.2 Events . . . 38

5.1.3 Verification Requests . . . 39

5.1.4 Storing of user approvals for identity verification . . . 40

5.1.5 Payment . . . 40

5.1.6 Storage of Hash . . . 40

5.2 Technical Aspects . . . 40

5.2.1 Compiling . . . 40

5.2.2 Deployment . . . 41

5.2.3 Testing . . . 42

5.2.4 Gas . . . 44

6 Evaluation **45**

6.1 Costs . . . 45

6.2 Waiting Time . . . 45

6.3 Scalability . . . 46

6.4 Security . . . 47

7 Summary and Conclusions **49**

8 Future Work **51**

Bibliography **53**

Abbreviations **57**

Glossary **59**

List of Figures **59**

List of Tables **61**

List of Listings **63**

A Installation Guidelines **67**

B Modification Scenario **71**

C Contents of the CD **75**

Chapter 1

Introduction

1.1 Description of Work

Financial institutions are obliged to know whom they are starting business relationships with, in order to exclude the possibility that the contracting party is involved in illicit activities such as money-laundering or funding of terrorist activities. Know your customer (KYC) have been developed to meet this requirement and have become indispensable in our modern world especially with the growing trend of blockchains. Blockchains are decentralized and distributed networks making controlling third parties unnecessary. Since transactions between peers can be anonymous and are based on trust in the consensus mechanism, KYC processes are more important than ever. KYC processes are however, both time consuming and expensive for the financial institution and their potential customer.

This thesis is about building a KYC platform, which can eliminate these inefficiencies. The platform provides customer identification processes as well as secure and verifiable data storage to guarantee that every person only has to identify themselves once. To do so, a secure and reliable web-based and open source registration platform will be designed and developed, to collect user data which is stored in a centralized database. The video identification must meet all FINMA requirements and contains a secure and interrupt-less video chat based on WebRTC.

Furthermore, the KYC platform will be integrated into the Ethereum network using smart contracts. These will be developed to handle authentication requests from external sources in an automatized and efficient way. Additionally, hashes of the user's data will be stored in the smart contract to prove the identity of the user in a verifiable and secure manner.

The main requirements for this platform are introduced briefly in the following paragraphs.

1.2 Blockchain and Smart Contracts

The Blockchain is a distributed digital and public ledger in a peer-to-peer (P2P) network that records transactions tamper-proof and autonomously without the need of the control by any single entity [5]. Through a decentralized consensus mechanism it is hosted by millions of nodes and is accessible to anyone on the internet. The blockchain was originally devised for the cryptocurrency Bitcoin but it has a lot more potential [6].
With this distributed ledger technology (DLT) all participants are able to communicate anonymously, but with proof of rights to interact, even if they do not trust or even know each other [32].

Creating and transferring money via Internet was originally the purpose of the DLT. Since the creation of the Bitcoin blockchain different purposes for the DLT have emerged, such as Ethereum. The Ethereum blockchain can run computer programs, so called Smart Contracts (SC) [32]. Smart Contracts are protocols which follow the ”if this then that” principle and are written as code into the blockchain allowing to trigger a contract clause automatically if the requirements are fulfilled. Smart Contracts can enhance efficiency in daily business, be it with purchase or rental contracts or electronic voting. Ethereum-based applications are called decentralized applications (dApps) [26].

Smart Contracts are implemented using a contract-oriented, high-level language called Solidity. Solidity's syntax is similar to the one of JavaScript and is designed for Ethereum [54].

1.3 Know Your Customer (KYC)

The Know Your Customer process has gained in importance since the emergence of anti-money laundering and funding of terrorist activity regulations and is a necessary basis for financial institutions to work with clients. Financial institutions are obliged to verify potential new customers to ensure that the customers aren't involved in above mentioned activities. The process starts, when a financial institution is approached by a potential customer, who intends to work with it. The customer will hand in basic identity information and documents which are used by the financial institution to check for illicit activities. This results in an internal document which serves as certificate that the KYC process has been properly conducted and confirms whether the application has been approved or rejected. This process is necessary every time a new potential customer intends to work with a financial institution. This leads to high costs on both sides [32].

1.4 WebRTC

WebRTC provides simple Application Programming Interfaces (APIs) for browsers and mobile applications with Real-Time Communication (RTC) capabilities. It is a free, open project whose components strive to enable rich, high-quality RTC applications which are

supported by browsers and platforms such as Chrome, Firefox, Android to name a few [61]. WebRTC eliminates the need of additional client software and plug-ins with simplifying voice, video, chat and data sharing (file sharing) for browser based communications [62].

1.5 Thesis Outline

Chapter 1 gives an overview of the description of work and explains the blockchain, know your customer (KYC) processes and WebRTC

Chapter 2 discusses the related work

Chapter 3 contains the design and requirements of the system

Chapter 4 is dedicated to the implementation aspects of the system. The first part describes which libraries were used to build the system. The second part discusses the concept of protection. In the third part, the login process is explained. Then, the complete video identification is graphically visualized and documented step by step. The last three parts talk about the KYC key validation, the Administrator page and the system architecture.

Chapter 5 outlines how the Smart Contract is implemented, how it was tested, deployed and integrated into the system

Chapter 6 evaluates the system's performance

Chapter 7 summarizes the project and conclusions are drawn

Chapter 8 explains how the system could improve with future work

Chapter 2

Related Work

In the financial services sector, performing Know Your Customer (KYC) verifications is essential to onboard new customers [39]. Since the common KYC process carries high costs for the financial institution and the potential customer, it is in the interest of both parties to make the process more efficient.
In this chapter companies are introduced, which have optimized the KYC-process to make them more efficient and to reduce costs and risks.

2.1 IDnow

IDnow is an identity verification platform which provides a wide range of KYC services which comply towards Anti-Money Laundering (AML) as well as the regulations of the corresponding authority. The goal is to provide tailored verification solutions especially in the financial sector but also in eCommerce, ICO & Crypto KYC. IDnow ensures security, legal compliance and a high quality of KYC processes [23]. Two of their identity verification methods are VideoIdent and AutoIdent.

VideoIdent: With VideoIdent a customer's identification can be verified quickly and comfortably using a P2P video chat with a responsible person. While video chatting the customer will be asked a few questions and asked to hold his/her ID into the camera of the laptop or mobile phone and tilt it so that the responsible person on the other side of the call can verify the ID [24].

AutoIdent: IDnow also provides a way to create your digital identity using artificial intelligence (AI) and Face recognition technology. The process is fully automated and there is no need for any ID verification staff [22].

IDnow provides an API with which a user identification process can be requested. To do so, IDnow needs the personal data of the user to be identified. To request an identification process, there are the following possibilities: REST API, Static link on IDnow, Web form or with a GET request. The results can either be downloaded or sent by email as JSON or as ZIP file [21].

2.2 CB Financial Services AG (CBFS)

Similar to IDnow, CBFS also provides a platform for online identification and conclusion of contracts. With the Identification solution CBFSIdent, the identification process is done via secure video streaming to verify identities and is provided as a "Software as a Service" (SaaS) or as a licence. Already onboarded customers are granted access to the CBFS's identification and signing service. The login credentials can also be integrated into a company, which allows the user to login directly into for example e-Banking services [8]. Raiffeisen, for example, adapted the technology of CBFS in 2016 to digitally onboard their customers [?].

2.3 Shufti Pro

Shufti Pro is a British company which provides efficient and accurate digital KYC verification services. It has the ability to verify people around the globe with their artificial and human hybrid technology within 60 seconds [50]. With face verification, document verification, hand written note verification and 2 factor authentication, Shufti Pro cuts costs and reduces risks at the same time [51].
There are two modes of identity verification with the Shufti Pro solution. When identitiy credentials from an end user for KYC is requested directly, it is called On-site VerificatTra-Tradion. Shufti Pro will then identify the user. The Off-site Verification mode is executed when identity verification of verifiable documents, provided by the customer, is requested. The benefit of the Off-site Verification is that there is no need for user-interaction [53].
Integrating Shufti Pro into your mobile applications, modules or online software allows to easily verify users. The possibilities are RESTFUL APIs, Android and iOS integrations or a hosted verification page [52].

2.4 Tradle

Tradle is an open-source platform allowing financial institutions to onboard their customers with blockchain-based bots. These automate the KYC processes for financial institutions to make them faster and more efficient providing security audits and building a trust provisioning network. Tradle's technology is integrated in the bank's web site. To use it, click on the link and download the App. In the App the client starts a chat with a bank's representative providing required information for the identification process, such as personal details, a selfie and snapshots of the identity documents. The bank's representative then verifies the data gathered and verifies the client. The information is then digitally signed and securely stored on the blockchain. If the client now would like to open an account at another bank, he can start a new chat within the app and share his verified data with them. The goal of Tradle's system is letting the client transfer trust, not assets, by ensuring that the transferred data is verifiable [12].

2.5 KYC-Chain

KYC-Chain enables businesses to handle the KYC processes for individuals and corporations more efficiently, using its B2B managed workflow application. By using the distributed ledger technology, the company ensures the privacy of individuals and corporations and their information, but still reaches a high level of transparency in the blockchain. The identity verification is accomplished with algorithmic validity checks of identity cards [27]. The platform ensures secure sharing of verifiable identity claims, data or documents. With KYC-Chain, the onboarding process of new customers can be carried out in a more efficient and trusted way for businesses and financial institutions [4].

2.6 KYCstart

KYCstart is a proof of concept (PoC) of the consultancy and accounting firm Deloitte providing KYC-as-a-service using blockchain technology. With "regulated KYC added-value service providers", Deloitte's blockchain-based solution could digitally perform KYC checks for clients who want to be onboarded by several financial institutions at once. The project is aiming for more efficient and cheaper customer onboarding by facilitating onboarding processes for financial institutions and creating digital identities for the customers, which can be shared among permissioned platforms and financial institutions. The benefits of the blockchain technology mean that customers and clients can control whom they want to share their company information with. Blockchain-based smart contracts enable them to keep track of the authorizations [28].

2.7 Comparison

Table 2.3 displays a comparison of the companies discussed in this chapter as well as the proposed method in this thesis with some selected features.
The first column shows that there are many ways to deploy KYC processes online. The auto identification requires technical tools such as AI and facial recognition which was out of scope for the platform developed in this thesis. Most of the established companies providing KYC-as-a-service don't use the blockchain technology. *Tradle* and *KYCstart* are aiming on having a digital identity of the institutional client on the blockchain and sharing the verified data among permissioned financial institutions. The goal of the method proposed in this thesis is to store user data on the blockchain and additionally to allow financial institutions to send verification requests through a smart contract.

Company	Identification	Blockchain	Access	Usage
IDnow	Video, Auto	x	API	Mobile/Web
CBFS	Video	x	SaaS or License	Web
Shufti Pro	Auto	x	API, Integration	Mobile
Tradle	Chat	✓	App, Web	Mobile/Web
KYC Chain	n/a	✓	Solution	Web
KYCstart	n/a	✓	n/a	n/a
Proposed Method	Video	✓	Smart Contract	Web

Table 2.1: Comparison of requirements

Chapter 3

Design

In this chapter, the design for the platform developed in this thesis will be presented. In Section 3.1, security requirements are identified and design ideas proposed to provide a secure and reliable platform. Then, the requirements for the video identification process regulated by FINMA are listed. Section 3.3 tackles the requirements of the smart contract to be developed for this system.

3.1 Security Requirements

A major security concern is the topic of *Cross-Site-Forgery-Requests (CSRF)*. Every since web applications with user authentication have been developed, this risk has to be handled with great care.
CSRF attacks are basically executed by tricking the browser to send unwanted HTTP requests. That means, that the attacker does not have to steal the user's identity, but only has to place an URL *e.g.* in an *img* tag which the user accesses on the malicious website, *e.g.* sending a request to delete the user's account [49].
To solve this issue, the use of *JSON Web Tokens (JWT)* is proposed. Using *JWT* allows us to use an *csrf token*. The server will require this token to validate any request. Setting a *csrf token* does not fully eliminate the risk of *CSRF* attacks. Cookies are required since the cookies are only accessible from the domain they were set and therefore making them unreadable for the malicious website. The *csrf token* will be stored in the cookie so that *CSRF* attacks will fail because they can not read the *csrf token* value and therefore are not able to complete requests [16].

3.2 Video Identification Requirements

For the video identification process, the platform must comply with the regulations FINMA has published in 2016 [14]. Hence, the technical aspects of the FINMA publication were considered carefully. In table 2.1 all relevant articles are listed with an abbreviation which will be referred to throughout the report.

Art.	Text	ID
11	”The financial intermediary structures the online onboarding process in such a way that the contracting party fills out the details required under Articles 44 and 60 AMLO-FINMA electronically and transfers them to the financial intermediary before the audio-visual identification interview takes place. [. . .] In addition, the financial intermediary checks the information gathered during the onboarding process against the information contained in the contracting party’s identification document.”	F-C1
12	”The financial intermediary must obtain the contracting party’s explicit consent to conduct the video identification and audio recording before starting the video interview.”	F-C2
13	”During the video transmission, the financial intermediary takes photographs of the contracting party [. . .].”	F-C3
14	”The financial intermediary also reviews the authenticity of the identification documents by using a machine to read and decrypt the information in the MRZ [. . .] The financial intermediary checks that the decrypted information matches the other information in the identification document and the data provided by the contracting party during the onboarding process.”	F-C4
15	”Only official identification documents issued by the relevant country can be accepted for this process. Moreover, the documents must have an MRZ and optical security features, such as holographic-cinematic marks or printed elements with latent image effects.”	F-C5
16	”The identity of the contracting party is to be verified by means of a TAN or another similar method.”	F-C6
17	”Each identification process must be documented. The photographs of the identification document and contracting party must be filed and archived along with the audio recording of the entire identification process..”	F-C7
18-21	”The financial intermediary will stop the video identification process if the picture or sound quality does not enable unambiguous identification of the contracting party, the financial intermediary identifies evidence of increased risks, or there are any doubts regarding the authenticity of the identification document or the identity of the contracting party.”	F-C8
24	”For contracting parties in the form of legal entities or partnerships, the financial intermediary must obtain an electronic extract either from the database of the relevant registration authority or from a trustworthy privately managed directory. The extract may also be submitted to the financial intermediary outside the video identification process.”	F-C9

Table 3.1: Requirements for video identification from FINMA Circular 2016 [14]

Before the video identification, F-C1 needs to be fulfilled. Table 2.2 describes these requirements [2].

Art.	Text	ID
44, 1a	When starting a business relationship with an individual, the financial intermediary collects the first name, last name, date of birth, address and nationality from the contracting party.	A-F1
44, 1b	When starting a business relationship with a legal entity, the financial intermediary collects the company name and the company address.	A-F2
44, 3	The financial intermediary takes note of the electronic copy of the power of attorney and checks the identity of the individuals representing the legal entity.	A-F3
60, 1	The written declaration of the contracting party about the beneficial owners must contain the first name, last name, date of birth, address and nationality.	A-F4

Table 3.2: Requirements for video identification from AMLO-FINMA [2]

To meet the requirements defined above, the following design proposals were made. For the requirements *A-F1, A-F2* and *A-F3* stated in table 3.2 a simple but still user friendly form could be implemented. The data would be stored in the database making it available for the users at any time. To prevent users from spaming the platform, email-based verification could be executed once the user registers proving that the user knows his email address and also that the user is in possession of the password to access the account.

For the video identification itself the usage of WebRTC APIs is proposed because the APIs provide secure and interrupt-less RTC capabilites. The security concern can be eliminated by using csrf tokens described in chapter 3.1 and a valid user id to authenticate. In this way, no unentitled user could access a channel. As described in *F-C6*, the platform could make use of the email-based verification.

Lastly, *F-C8* is tackled with the following design proposal. The administrator should be able stop the video transmission if there are any doubts regarding authenticity of the identification document or the identity of the contracting party.

3.2.1 Smart Contract

The smart contract which has to be developed brings requirements such as accessibility, anonymity and usability. The accessibility requirement for users could be met by having a browser extension installed allowing the user to create a new wallet. Since sending transactions to the Ethereum network always cost an amount of Ether, it is important that the browser extension has access to the user's wallet to transfer the funds.

Blockchain transactions are transparent and visible for all users in the network. Since the goal of [15] is to allow user's complete anonymity, it must be considered to make smart contract requests anonymous. A design proposal is to use a unique hashed key (*KYC key*) with which users could register themselves at connected platforms anonymously while providing consent that the user's data could be shared with the platform. To prevent that the *KYC key* is copied once a smart contract request was processed, it is suggested that the mentioned key be hashed.

As mentioned above, each transaction made to the Ethereum network requires a small payment of Ether amount. This amount is called *gas*. Another aspect, which has to be considered, is the block time of the Ethereum network. It takes some time to process the transaction because every transaction has to wait for a new block to be created. In a system enabling user-interaction with the blockchain this can decrease the usability due to long transaction times. If the amount of Ether the user is willing to pay per unit of *gas* increases, it could ensure that the transaction is mined quicker. Unfortunately, this results in a trade-off between costs for transactions, the so called *gasPrice*, and block time of transactions meaning the usability decreases when the costs are reduced and vice versa. Therefore, the goal is to find a well balanced *gasPrice*.

Chapter 4

Implementation

This chapter explains and demonstrates how the technology behind the KYC platform is built up and how the functions were implemented.

Three categories of users exist in the system. The administrator acts as financial intermediary and performs the video identification. He also has the rights to see all the users' data. Then, there is a user who has successfully completed the KYC-process. For simplicity, this user will be referred to throughout this chapter as "identified user". The last user has not yet gone through the KYC-process and is referred to as "unidentified user". The following sub-chapters describe the libraries and the main features of the system as well as the Smart Contract. They also explain the activities of server and client.

4.1 JavaScript libraries

In this section all relevant JavaScript libraries used in the client and server are presented and explained.

4.1.1 Server

Express: Express is an easy and flexible framework of web applications for Node.js. It is needed to submit HTTP requests and responses in the Server [20]. Express also provides a range of Middlewares, i.e body-parser, which is used to parse request bodies sent by the client [40] as well as cookie-parser, which helps to parse cookie headers to get information stored in the cookie [19].

Next.js: The server is built with the React framework Next.js. Next.js allows server-side rendering of a React App and provides easy and convenient dynamic routing [35].

MySQL: The server is connected to a local MySQL Server which is used as database. The Mysql driver "mysql" is needed to access the database from the Node.js application [59].

Nodemailer: The Nodemailer module enables the easy sending of emails from the Node.js application [38]. In this platform it is especially used to send email verification links, password reset links and one time passwords (OTP).

Bcrypt: Bcrypt is a Node.js library to hash passwords and also to compare already existing hashed passwords [7]. It is used to store the users password as a hash in the database to prevent theft. The compare function is needed for the login process.

Crypto: Crypto is a built-in module, which brings a set of cryptographic functionalities. For example it allows the generation of hashes from plain text [37]. Crypto is utilized to hash the JSON web tokens.

JSON Web Token (JWT): A JSON Web Token is a secure JSON object which represents information between two parties. A JWT is used for authorization, which is the most common usage, and information exchange. The JSON Web tokens are composed of Header, Payload and Signature [25].

4.1.2 Client

React: React is a very convenient, component-based JavaScript library for building user interfaces [46]. With React, JSX was used to build the UI. JSX is a syntax extension to JavaScript to define how the user interface should look like [45].

Web3.js: Web3.js is an Ethereum JavaScript API which uses a HTTP or IPC connection to enable interaction with a local or remote ethereum node [60].

Axios: Using axios simplifies the sending of asynchronous HTTP requests from the Client side of the platform [3]. It is used equivalent to Express, which handles the HTTP requests in the Server.

Simple-peer: Simple-peer is an API for WebRTC, which provides simple video, voice and data channels. It is used to connect two peers automatically by exchanging the offer of the initiator and the answer from the receiver to establish the connection. However, to automatize this process between two browsers a signaling server is used [1].

Semantic Ui React: This Semantic UI integration has the purpose of building and styling the user interface. Semantic UI React helps to build in easy to use elements, views, modules, behaviours and addons in a user-friendly way [48].

SweetAlert2: Sweet Alert 2 provides a very easy way to show alerts in the browser. The alerts are customizable but still very easy to use [56]. In this application, Sweet Alert 2 is used to show success messages to the user to bring a better understanding when a process has ended and the system is ready to proceed.

4.1.3 APIs

Pusher: Pusher provides a set of APIs to build realtime applications [42]. The Pusher channels allow the setting up of a connection between servers, apps and devices to establish realtime communication and is used for the video chat in the KYC-platform [44]. Pusher serves as signaling server, which exchanges the signaling data from simple-peer between two peers until a connection is established [1].

4.1.4 Browser Extensions

MetaMask: To interact with the Ethereum network and the smart contract, the administrator and the users must have installed the MetaMask extension on the browser. MetaMask runs on *Google Chrome*, *Firefox*, *Opera* and *Brave Browser* and serves as a bridge between the Ethereum network and the browser [30]. The user has to create a new wallet with an existing account or a new account. The reason why MetaMask is needed is that the browser has to have access to the Ethereum wallet to be able to interact and transact with the smart contract.

4.2 Concept of protection

In this section the following topics are discussed: How are the pages of the platform protected from unauthorized users? and how can it be ensured that the user is the owner of the provided email address? Furthermore, the aspects a of secure password reset functionality are also considered. Lastly, the strength of the user passwords is discussed.

The following protection functions are basically all executed server-side.

4.2.1 Protection of pages

Since the system contains confidential data, it is important to protect the pages from unauthorized users. Therefore a function was implemented which verifies whether the current user is authorized to access the protected pages. Almost all pages are protected except for the login, register, password reset and the KYC key validation page.
To verify the user, JSON Web Tokens (JWT) are created after login (chapter 4.3). If the user has not logged in, no cookie is created and thus the user cannot access the protected pages. The server verifies the user and checks if he is authorized to access the targeted page.

This is also the case when a non administrator tries to access the admin page. The code which is used in this case is illustrated in listing 4.1. It is possible to store information in these tokens which can be decoded with a secret. Before this code gets executed, a SQL query is executed to check what kind of category of user the system is dealing with. In this example it is an administrator. Then, a token is assigned to that specific user. The

token will store the username in the header, which is used to identify the user at any time (line 3). The role is set to 1 that means the user is an administrator (line 4). Also, the *reg* measure is set to 1 to prove the user has completed the KYC process (line 5). In the end, a *csrf token* is added to prevent cross-site request forgery (CSRF) attacks (line 6). Storing this *csrf token* in a cookie as will be discussed in chapter 4.3.
Line 10 is very important in which a secret is attached to the token as well as an expiry time. The secret is used to decode the token and should not fall into the wrong hands.

```
1  let adminToken = jwt.sign(
2    {
3      username: body.username,
4      role: 1,
5      reg: 1,
6      csrfToken: crypto
7        .createHash("sha256")
8        .update(body.username)
9        .digest("hex")
10   } ,secret, {expiresIn: 36000}
11 );
```

Listing 4.1: Signing a JWT

Since the token is assigned to the user we need a server function which verifies the token and redirects the user to the correct page, as shown in listing 4.2. Once a user is trying to access the administrator page, the server will check the request with the function on line 1. It will load the protectAdminPage function before the routing is done.
The function protectAdminPage checks whether the user is authorized to access the administrator page. First of all, it checks whether there is a cookie stored in the browser (line 6). If there is no token, the user is not logged in and will be redirected to the login page (line 20). If there is a cookie available, the *jwt.verify()* function decodes the token with the secret and checks if the role of the user is 1, which means that the user is an administrator (line 12). If the user is no administrator he will be redirected to an error page, otherwise the route to the path */admin* is loaded.

```
1  server.get("/admin", protectedAdminPage, (req, response, next) =>
     {
2    return next();
3  });
4  [...]
5  function protectedAdminPage(req, res, next) {
6    const token = req.cookies["x-access-token"];
7    if (token) {
8      jwt.verify(token, secret, (err, decoded) => {
9        if (err) {
10         console.log(err);
11       } else {
12         if (decoded.role !== 1) {
13           res.redirect("/error");
14         } else {
15           return next();
```

```
16          }
17        }
18      });
19    } else {
20      res.redirect("/login");
21    }
22  }
```

Listing 4.2: Verifying tokens

If a user has not logged in it means that there was no token signed and therefore no cookie available. To prevent a non logged in user to access protected pages, a similar function to the one above was implemented. The only difference is that it will not check the users rights but will check if there is a user token available in the cookies. If not, the user will be redirected to the *login* page when trying to access the protected pages.

4.2.2 Email verification

Another aspect of protection is the email verification. Since the processes done in the system cost money and time, there is no interest for people to spam the system. Therefore, the email verification will help verify entered email addresses and prevent multiple usages of an email address as described in listing 4.3. When a user initializes a registration and fills out the form, the data will be stored in the database, but still set as inactive since he has not yet confirmed his email address which means the user can not log in. Once the data is stored in the database, the *jwt.sign()* function is executed (line 1). The first step is to sign a token with the user name stored in it (line 3). Here, we use the Nodemailer library for the first time. On line 13 a function is created to send an email to the user with a link. For verification purposes the just created *emailToken* will be attached to the link (line 14). This is important, because once the link has been accessed, the server gets the route and verifies the token stored in the link. If the token is valid, the database column *active* is set to "1" and the user is free to proceed and login.

```
1  jwt.sign(
2    {
3      username: body.username,
4      emailToken: crypto
5      .createHash("sha256")
6      .update(body.username)
7      .digest("hex")
8    },
9    EMAIL_SECRET,
10   {
11     expiresIn: 36000 //1h
12   },
13   (err, emailToken) => {
14     const url = `http://localhost:3000/activate/${emailToken}`;
15     transporter.sendMail({
16       from: "no.reply.sealle@gmail.com",
```

```
17        to: body.email,
18        subject: "Confirm Email",
19        html: `Please click this link to confirm your email:
20        <br/><a href="${url}">${url}</a>`
21      });
22    }
23  );
```

Listing 4.3: Sending email verification links

Figure 4.1 shows an example of a email confirmation link containing the domain of the system, the endpoint *activate* and the hashed token.

http://localhost:3000/activate/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IlNlYmFzdGlhbi
IsImVtYWlsVG9rZW4iOiJkYjAyODRmYzgwYjQ4ODFmYTU0NTk1MzQ0MWEyMT
k1ODQ3MGM0OWQ3MzUwODcxNGE4NGEyNTljYjYyYTVkYmI1IiwiaWF0IjoxNT
Q4MjM1MzE4LCJleHAiOjE1NDgyNzEzMTh9.Xi0a2rl8PTJCzxzt9qKKauN_lyy10ch5Vlq2pJYFJ4M

Figure 4.1: Email confirmation link

4.2.3 Password Reset

The struggle of forgetting a password for a specific login is well known. Therefore it is always good to have a reset password function. As developer it is crucial to implement this correctly otherwise the system will lack in security.

The approach to create a password reset is described below:

1. Create a page where the user must enter his user name and email address.
2. Receive the user name and email address and create a token containing the user name. That token is placed on the link which is sent to the corresponding email address. The link expires after 30 minutes.
3. Once the link is activated by the user, the server verifies the token and redirects the user to a new page on which the user can enter a new password. This page is only accessible with a valid link respectively valid token.
4. By hitting submit, the Client sends the new password to the Server which updates the password in the database.

The code for sending the link to the email address is similar to listing 4.3 and the password reset link looks similar to the one illustrated in figure 4.1 except for the endpoint.

4.2.4 Password Strength

For the user it is crucial to secure their personal data from unentitled persons. Using strong passwords can prevent users to be hacked. In this platform the user must have a password with the following characteristics: The password must contain at least 1 upper case letter, 1 lower case letter, 1 digit, 1 special character and has to have a length eight characters.

4.3 Login

The login is one of the most important features in the system. Here the tokens are signed to the specific users. For each user there is a specific token saved in the cookies. To know which user has which role, there are two columns in the database: the first one is *privileges* which is per default *user* and *isRegistered*, which shows if the user has completed the video identification process or not.
An overview of the user categories and the measures are listed below:

- Administrator: Privileges=*"admin"* and isRegistered=*"yes"*
- Identified User: Privileges=*"user"* and isRegistered=*"yes"*
- Unidentified User: Privileges=*"user"* and isRegistered=*"no"*

If tokens are created it has to be ensured that every user has the correct one because the token will store the role of the user and the user name to protect the pages the users are not authorized to enter. This topic was discussed in chapter 4.2.1.

Server: After submitting on the client-side, the server receives the user name and the password. The password will be checked with a *Bcrypt* compare function. Also the user name will be verified. If the user name and password match, the server creates the specific token described in 4.2.1. The Server then sends back the token and the role of the user.

Client: Similar to the registration, the user enters his user name and his password. This is only possible once the user has verified his email address. After submitting the client will make an *axios.post()* to the server with the form data attached to it. Again, the response will be either an error message or a success message. If the login was successful, the client receives the tokens previously created by the server and the role of the user.

This is an example on how cookies are created and the tokens placed into it. Listing 4.4 shows how a cookie is set to the browser for a administrator token. In this case, the client has received the register status, which is basically the value of the *isRegistered* database entry and the *privilege*. These are used to check which category the user belongs to as seen on line 1. Line 2 of the listing illustrates how the cookie is created and the token is placed into it. The *setCookie()* function is a JavaScript function which can be used to create cookies [58]. The first parameter of the *setCookie()* function is the cookie name. Here the name is *x-access-token* because the token is used to grant access to the system.

The second parameter is the token value, which has been sent by the server. The last parameter is the expiration date defined in days. This means the cookie will last for one hour in the browser.

```
1 if (res.data.registerStatus == "yes" && res.data.privileg == "
     admin") {
2   setCookie("x-access-token", res.data.adminToken, 1);
3   Router.push("/admin");
4 }
```

Listing 4.4: Setting cookies into the Browser

4.4 Video identification process

For the video identification process, all requirements of the FINMA circular 2016 described in chapter 3.2 have to be met. Figure 4.2 shows each step a user has to go through to claim his KYC key. These tasks are described and commented step by step in the following subsections.

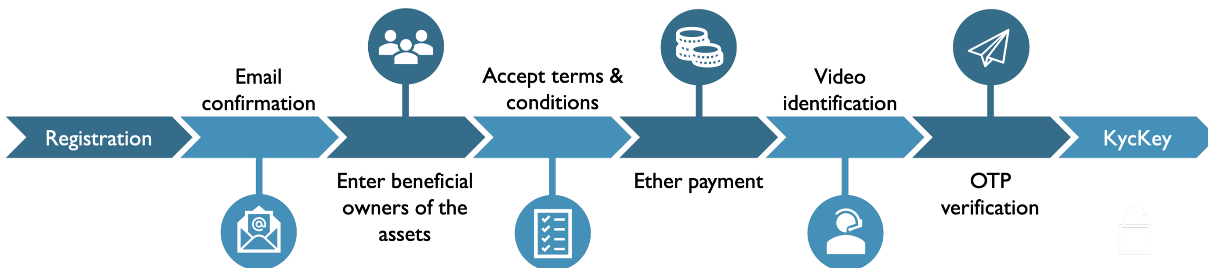

Figure 4.2: Video identification process

4.4.1 Registration

A fundamental feature of the system is the registration. The goal is to collect the required data of the user and store it in the database. On the registration page a person is asked to fill in his personal information (*A-F1*) and additionally to upload pictures of the back- and front-side of his identity card. If this person enters on behalf of a company, he will need to hand in additional information as required from *A-F2* and *A-F3*. Also, to simplify *F-C9*, this system requires the representative of the legal entity to deliver the electronic extract of the relevant registration authority. After successful submission the person will receive an email with a confirmation link. Chapter 4.2.2 describes how this is accomplished.

Client: On client-side, the user enters the required information. This information will be saved in a form data object when submitted. All information is required and the submission is not possible without uploading two pictures (back- and front-side of identity card).

In order to send the form data object to the server, an asynchronous *axios.post()* request directly to the register endpoint will be executed (listing 4.5 line 1). The *await* keyword basically makes the client wait until the server responds to the request before the following code is executed. The response from the server will either be a success message or an error message, depending on the verification check in the server. This is done for every request from the client side but with different endpoints.

```
1 const response = await axios.post(window.location.origin + "/
     register", formData);
```

Listing 4.5: Client-side server request

Figure 4.3 shows how the registration form looks like and which information is required.

Registration

Person Company Back to Login

Username * Password *

Username Password

First Name * Last Name *

First Name Last Name

Street * House Number *

Street House Number

Postal Code * Place of Residenz *

Postal Code Place of Residence

Date of Birth * Nationality *

DD.MM.YYYY e.g. Switzerland

Email * Mobile Number *

email@email.com +4* ** *** ** **

ID Number * ID Type *

ID Number Choose a ID Type

ID front- and backside! (.png or .jpg) *

Choose Files No file chosen

Submit

Figure 4.3: Registration Form

Server: Similar to the Client, the server makes an HTTP request with Express to the */register* endpoint. When the server receives the form data from the client it first makes a database check to see whether the user name or the email address already exist. If yes, it responds with an error message to the client. If no, the *Bcrypt* library hashes the received password and then all data is stored in the database. In the end a JSON Web Token is created and an email is sent to the received email address with a link for email verification, in which the token is stored as seen in chapter 4.2.2. If everything runs successfully, the server will pass a success message to the client.

4.4.2 Email Confirmation

The next step of the identification process is to confirm the email address. This topic was discussed in chapter 4.2.2. After successful registration, the user receives a notification which says that an email was sent to the user with an email confirmation link as illustrated in figure 4.4. The user can confirm or request to resend the email. Once confirmed, the user is redirected to the login page where he will log in and proceed with the next step.

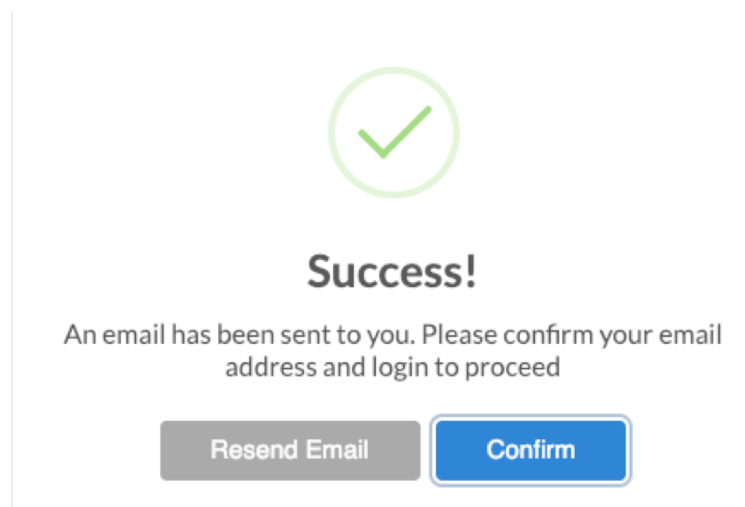

Figure 4.4: Success Message

4.4.3 Beneficial Owners

After the user logs in, he will need to declare if there are beneficial owners on his assets as described in *A-F4*. In the system, this requirement was implemented using simple check boxes. The user has the possibility to declare that he is the sole beneficial owner of his assets which will redirect him to the next page. If the user has beneficial owners a new form opens which has to be filled in. There is the possibility to enter the data of three beneficial owners, but only one is mandatory. In the case that the user has beneficial owners, a server request is made from the client sending the data of the beneficial owners. This data is then stored in the *beneficialOwners* database table along with the user name. Figure 4.5 illustrates this form.

Figure 4.5: Enter beneficial owners of the assets

4.4.4 Terms & Conditions

For the next step the user must agree to the terms and conditions, some of which are required by FINMA. For the first two points the user has to agree to the video identification as well as the recording of the audio line (*F-C2*). To enable a good quality in the video chat, the user has to confirm to have his identity card ready, have a good internet connection, to be in a silent environment and have a good microphone quality. The last agreement is that the user will pay an amount of Ether for the video identification service. This will be discussed in the next section. The user can only proceed if he has checked all boxes as illustrated in figure 4.6.

Figure 4.6: Terms and Conditions

4.4.5 Ether Payment

The first and only time a user interacts with the blockchain is when he is asked to pay an amount of Ether to the administrator address. When the user accesses the Ether payment view he will be asked to log into MetaMask (chapter 4.1.4) if he is not yet logged in. It is not possible to enter negative values nor to leave the input field blank. Once a value is entered, MetaMask will ask to confirm the transaction. After the user has confirmed, the transaction will take some time. If the transaction was successful, the user is redirected to the *videochat* page.

In listing 4.6, a part of the client-side code regarding the Ether payment is illustrated. First of all, the *web3* has to get the user's account (line 3). Then, the contract function *payKYC()* is executed and defined from which address the transaction is being sent and the value of Ether, which was entered by the user (line 6 to 9). The smart contract function is explained in chapter 5.1.4. If the transaction was successful, a *Swal* (see chapter 4.1.2) alert will show up with a message and a button (line 10). The button has it's own function, which fetches a new user token from the server, similarly to chapter 4.3 (line 16). The reason behind this is to prevent users accessing the *videochat* page without completing the tasks of entering the beneficial owners, agreeing to the terms and conditions and also the payment. If the server response is a success, a new cookie is set to the browser and the received token placed in it (line 21). In the end, the user is redirected to the *videochat* page.

```
1 submit = async () => {
2     //get eth accounts
3     let accounts = await web3.eth.getAccounts();
4 [...]
5     //execute payment to admin address
6     await contract.methods.payKYC().send({
7         from: accounts[0],
8         value: web3.utils.toWei(this.state.etherValue, "ether")
9     });
10    swal({
11        title: "Thank You!",
12        text: "You will be redirected to the video identification"
          ,
13        type: "success",
14        onClose: async () => {
15            //get videochat token from server to store in cookie
16            let response = await axios.post(
17                window.location.origin + "/clickandpay"
18            );
19            if (response.data.success) {
20                console.log("success");
21                setCookie("x-access-token", response.data.
                      videoCookie, 1);
22                Router.push("/videochat");
23            }
24        }
25    });
26 [...]
27 };
```

Listing 4.6: Ether payment

4.4.6 Video Identification

The video identification is a key feature in the system and is done with a video chat based on WebRTC. The video chat must be secure and interrupt-less. To fulfill these requirements the Pusher API and Simple-Peer were used. This section describes how the administrator and the users access a channel and how the connection is established between them.

First, a *MediaHandler.js* file was implemented to let the browser ask for access to the user's camera and the microphone when they access the video chat. The *MediaHandler* and the core functions of the video chat were adopted from [13]. Then, when users or the administrator access the video chat, they first need to be authenticated by the Pusher API. The authentication process runs automatically in the background so that there is no need for the users to take action. After successful authentication, which requires a *csrf token* and the user name, the users and the administrator are subscribed to the channel.

It has to be noted that the administrator has to always be the first to connect the channel and will not disconnect while in an active connection. Once subscribed to the channel, the administrator will always be subscribed even if he is looking at a user profile. This is necessary to receive notifications when a new user has subscribed to the channel.

After subscribing to the presence channel, a client event will be bound to the channel. Client events enable a connection directly via the socket without needing to use the server [43]. in listing 4.7 on line 1 the client signal is bound to the pusher channel. Then, a new peer is defined as the user which will receive the first signal, or offer (line 2). This user will not be an initiator because he will enter the channel first and won't have a peer at this time. In this application, the user is always the called one. Visible on line 6, *Peer.signal(signal.data)* is used to establish the peer-to-peer connection and serves as an answer which will be sent as soon as the *client-signal* is triggered (listing 4.8 line 8).

```
1 channelName.bind(`client-signal-${this.currentUser.id}`, signal =>
    {
2   let peer = this.peers[signal.userId];
3   if (peer === undefined) {
4     peer = this.startPeer(signal.userId, false);
5   }
6   peer.signal(signal.data);
7 });
```

Listing 4.7: Bind client event to the Pusher channel

To start a call, the administrator will execute the *callTo()* function. This function first executes the *startPeer()* function where the *userId* is the connected user in the channel (line 5). Then, an *axios.post()* is triggered to get the peer's data from the database, such as the images of the identification documents (line 9). At the same time, the *peer.on('signal')* function beginning on line 30 triggers the predefined client signal (line 31) and sends data, which is an offer, to his peer (line 34). As already explained, the *signal.data* will return as an answer and the connection is established. Further, with a *peer.on('stream')* function, the media stream of the peer is fetched to enable the video and voice transmission.

```
1 //triggers the call to peer
2 callTo = async userId => {
3 [...]
4     //start call to userId
5     this.peers[userId] = this.startPeer(userId);
6     let currentUser = userId;
7     console.log(userId);
8     //get data of the peer
9     let response = await axios.post(
10      window.location.origin + "/videochat/user",
11      {
12        currentUser
13      }
14    );
15    if (response.data.success) {
16      this.setState({
```

```
17          usrs: response.data.userData,
18          img1: response.data.img1,
19          img2: response.data.img2
20        });
21      }
22  };
23  [...]
24  startPeer(userId, initiator = true) {
25    const peer = new Peer({
26      initiator,
27      stream: this.currentUser.stream
28    });
29
30    peer.on("signal", data => {
31      channelName.trigger(`client-signal-${userId}`, {
32      type: "signal",
33      userId: this.currentUser.id,
34      data: data,
35      });
36    });
37  [...]
```

Listing 4.8: Establish P2P connection

At this time, the video and audio transmission has started and the administrator and the user can communicate face to face. To help to describe the following FINMA requirements, figure 4.7 shows what the administrator console looks like. On the left side, the video transmission takes place. Beneath the video container there are three buttons, all serving a different purpose.
As required in *F-C3*, the button on the left enables the administrator to take a photograph of his counterpart. As soon as the snapshot is taken, the image will be sent to the server and stored in the database.
The middle button is used to approve the user and send an OTP to him. The button on the right stops the call. Both of these functions are described in chapter 4.4.7. Next to the stop call button, there is a spinning icon, which means the audio line of the transmission is being recorded.
The button at the bottom of the page is the call button. For every user waiting for the administrator to call them, there is a button with the user name on it. Thus, the administrator can choose who he will call.
On the right side of the page, the identification documents of the user appear as soon as the video transmission started. This should cover the requirements *F-C4* and *F-C5*. Since for registration, only the identity card and the drivers license are allowed on this platform, *F-C5* is covered. The functions to comply with the requirement *F-C4*, are illustrated in listing 4.9.

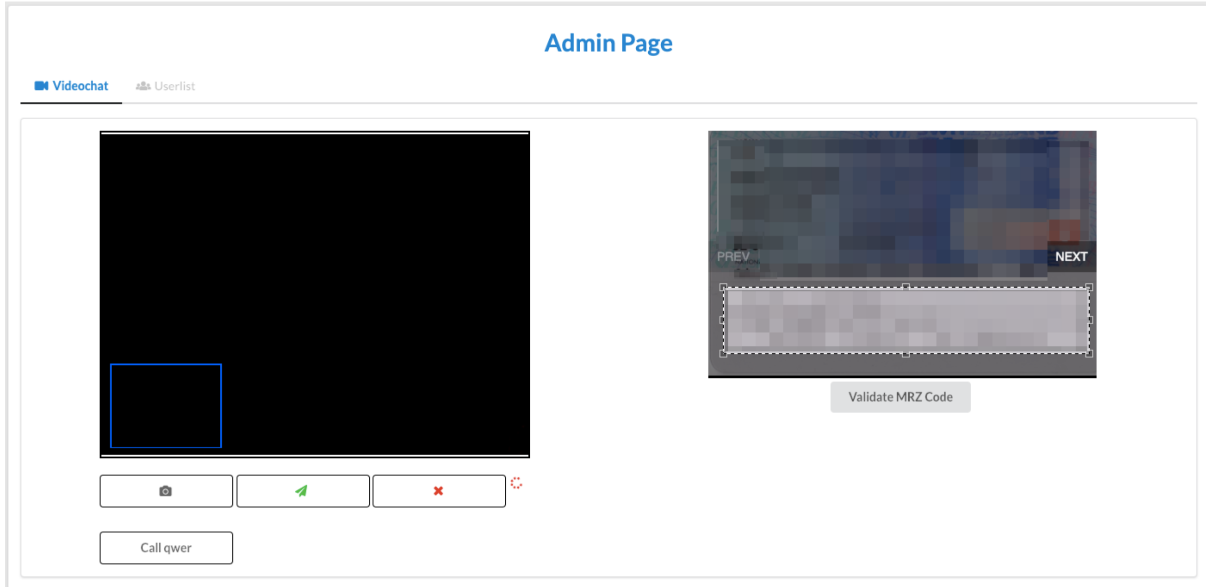

Figure 4.7: Administrator page

To perform the OCR scan the *tesseract.js* library from Project Naptha was used [41]. For the MRZ scan the *mrz* package for npm was required [9]. As mentioned above the administrator can crop the image with a selection on the identity card. By then clicking on the *Validate MRZ Code* button, the *ocrScan()* function is triggered. This function first gets the cropped image from the canvas element (line 7) and forwards it to *tesseract*, which then recognizes the charactes (line 9). This process can take a while depending on the image quality. The *mrz* library needs a specific input, that's why the output of the OCR scan needs to be prepared. Since the MRZ codes of identity cards always have the same amount of characters, the *tesseract* output can be divided into predefined substrings (line 17). The same procedure is done for the drivers licence, where the amount of characters differ from the ones on identity cards (line 23).
Once the array is prepared, it is read by the *mrz* library (line 29). The result of the MRZ validation is a boolean. Accordingly a success or error message will be shown to the administrator.

```
1 //make optical character recognition (ocr) on the mrz code of the
      identity card
2 //prepare output for mrz scanner
3 //scan the mrz code and verify it
4 ocrScan = () => {
5     this.setState({ loadingOCR: true });
6     //get cropped image from canvas
7     let image = document.getElementById("mrz-code");
8     //use tesseract to read the mrz code
9     Tesseract.recognize(image).then((result, err) => {
10    if (err) {
11      console.log(err);
12    }
13    let ocrLines = [];
14    console.log(this.state.usrs[0].idType);
15    if (this.state.usrs[0].idType == "id") {
```

```
16        //prepare data for identity card mrz verification
17        let ocrText1 = JSON.stringify(result.text).substring(1,31);
18        let ocrText2 = JSON.stringify(result.text).substring(33,63);
19        let ocrText3 = JSON.stringify(result.text).substring(65,95);
20        ocrLines.push(ocrText1, ocrText2, ocrText3);
21      } else {
22        //prepare data for drivers license mrz verification
23        let ocrText1 = JSON.stringify(result.text).substring(1,10);
24        let ocrText2 = JSON.stringify(result.text).substring(12,42);
25        let ocrText3 = JSON.stringify(result.text).substring(44,74);
26        ocrLines.push(ocrText1, ocrText2, ocrText3);
27      }
28      //verify mrz code
29      let res = parse(ocrLines);
30      if (res.valid === true) {
31        this.setState({ idIsValid: true });
32      } else {
33        this.setState({ idIsValid: false });
34      }
35  [...]
36    });
37  };
```

Listing 4.9: Identity Card Validation

4.4.7 OTP Verification

As the video identification process is coming to an end, the administrator has to decide if the user will be approved or declined. If the user is approved, the administrator triggers a function, which sends an One Time Password (OTP) to the user's email via server. *F-C6* describes this is requirement. For this functionality, a library called otplib, which allows to generate OTPs, was used [63]. The OTP is sent to the server, where the OTP is stored along with the user name in the database and then is sent via email to the user's email address. Once received, the user enters the OTP in the designated field. If entered correctly the administrator will receive a notification in form of a popup.

Client: For this functionality a new client event had to be bound to the channel as shown in listing 4.10 (line 2). This event is triggered for the administrator as soon as the user enters the correct OTP and submits. As described, the administrator will receive a *swal* alert containing a message and a button with a function (line3). Clicking on the button executes a row of functions. First, the audio recorder will be stopped (line 12) and the recorded *Binary Large Object (Blob)* will be converted into a *wav* file for the server to read (line 18). The file is stored in a form data object along with the user name and the file name and sent to the server to be stored in the database. This is a FINMA requirement described in *F-C7*. When the client receives a success from the server, the user's first name, last name, id number and the KYC key will be hashed with the *web3.utils.soliditySha3()* function (line 39) and sent to the smart contract (line 42). The smart contract function is explained in chapter 5.1.5.

```
1 //listener for admin if user has successfully entered otp and
    ended the process
2 channelName.bind(`client-approval-${this.currentUser.id}`, message
    => {
3     swal({
4       title: "Success!",
5       text: "The user has entered the correct OTP",
6       type: "success",
7       confirmButtonText: "Confirm",
8       //when admin confirms, send audio recording in db
9       //hash user data and send to smart contract
10      onClose: () => {
11        //Stop audio recording and send .wav file to server
12        recordRTC.stopRecording(async () => {
13          let formData = new FormData();
14          let recordedBlob = recordRTC.getBlob();
15          console.log(recordedBlob);
16          let fileName = `${userName}.wav`;
17          //convert recorded blob to a file and send it to the
              server
18          let file = new File([recordedBlob], fileName, {
19            mimeType: "audio/wav"
20          });
21          console.log(file);
22          formData.append("fileName", fileName);
23          formData.append("file", file);
24          formData.append("userName", userName);
25          this.setState({ isRecording: "" });
26          //get user data from server
27          let response = await axios.post(
28            window.location.origin + "/approval",
29            formData
30          );
31          if (response.data.success) {
32            let accounts = await web3.eth.getAccounts();
33            let fname = response.data.fname;
34            let lname = response.data.lname;
35            let idNum = response.data.idNum;
36            let kycKey = response.data.kycKey;
37            console.log(fname, lname, idNum, kycKey);
38            //hash user data and store in smart contract
39            let hash = web3.utils.soliditySha3(
40              `${fname} ${lname} ${idNum} ${kycKey}`
41            );
42            contract.methods.storeHash(hash).send({
43              from: accounts[0]
44            });
45          } else {
46            console.log("error");
```

```
47              }
48            });
49          }
50        });
51      });
52  };
```

Listing 4.10: OTP verification

Server: On the server side, the data is stored in the database but also a unique KYC key for the user is created (lisiting 4.11). This was done with the function *web3.eth.accounts.create()* (line 8). This function creates a new Ethereum account and the account address will be assigned to the user as KYC key. Also, on line 11, the KYC key needs to be hashed and stored with the KYC key to compare with the hash the external platforms send to verify as explained in chapter 5.1.3. The database columns *kycKey, isRegistered, audio, kycHash* and *otpToken* are updated in the user's database table (line 15). On line 20, a new query is defined which gets the *first name, last name, idNum* and *kycKey* to send them back to the client, where they are hashed and stored on the smart contract (line 39 to 42). Once the *database.connection.query()* function with the *storeKycKeyUsers* query is executed, the audio file fetched on line 6 will be stored in the *static* directory (line 29). In the end, the *getHashInfo* query is executed to retrieve the previously mentioned data (line 33).

```
1 // when otp verified, create and store kycKey in DB
2 server.post("/approval", urlEncodedParser, (req, response) => {
3     let body = req.body;
4     let userName = body.userName;
5     let audioName = req.body.fileName;
6     let audio = req.files.file;
7     //create new kycKey
8     let newAccount = web3.eth.accounts.create();
9     let newKycKey = newAccount.address;
10    //hash the kyc key
11    let kycHash = web3.utils.soliditySha3(newKycKey);
12    //insert kycKey in DB and set user to isRegistered
13    //set otpToken to null after verification
14    let storekycKeyUsers = SqlString.format(
15    "UPDATE users SET kycKey=?, isRegistered=?, audio=?, kycHash
        =?, otpToken=? WHERE username=?",
16    [newKycKey, "yes", audioName, kycHash, null, userName]
17     );
18     let getHashInfo = SqlString.format(
19     "SELECT fname, lname, idNum, kycKey FROM users WHERE username
         =?",
20     [userName]
21     );
22 [...]
23     database.connection.query(storekycKeyUsers, function(err, res
         , fields) {
24         if (err) {
```

```
25          throw err;
26        } else {
27          //store audio file in static folder
28          audio.mv("static/" + audioName, function(err) {
29            if (err) console.log(err);
30          });
31 [...]
32          database.connection.query(getHashInfo, (err, res, fields
               ) => {
33            if (err) {
34              throw err;
35            } else {
36              response.status(200).json({
37                success: true,
38                fname: res[0].fname,
39                lname: res[0].lname,
40                idNum: res[0].idNum,
41                kycKey: res[0].kycKey
42              });
43            }
44          });
45        }
46      });
47 });
```

Listing 4.11: Create the KYC key

This is the use case if the user was approved. However, there has to be a functionality to stop the video identification (*F-C8*). If the administrator has any doubts in terms of the identity or the authenticity of the identification document of the user, the administrator has a button to stop the call immediately. When the call is stopped, the user will be redirected to the *login* page and his cookie is removed from the browser to ensure that he has no more access to the protected platform pages.
Another thing that happens in the system when a call is stopped, is that the user data, including the snapshot and the audio recording, is sent to the server and stored in the database similar to the OTP verification process described in chapter 4.4.7. The only difference is, that in the server no KYC key is created but the *kycKey* column is updated with *declined*. A user having this in the database will be blocked from using this account further. This means that the account is blocked for the user and not accessible anymore.

4.4.8 Profile

If a user has successfully accomplished the KYC-process, he will be able to access his profile. The video identification process ends here as the user can access his KYC key. With this key he will be able to register anonymously at an external platform.

4.5 KYC key validation

To validate the KYC key at any time even if the database of the platform is down, users can access the *validation* page on the platform and insert the required data into the designated fields. The system will then fetch an array from the smart contract to check if the KYC key is valid. The smart contract function is explained in more detail in chapter 5.1.6.

Listing 4.12 shows how this function works. First, the *handleFormSubmit()* function gets the user inputs (line 4 to 5). Then, the function *getHashes()* from the smart contract, described in chapter 5.1.6, is called (line 10). The result is an array of all hashes stored in the smart contract (line 13). To compare the user inputs with the hashes in the array, the user inputs, *fname, lname, idNum* and *kycKey* are hashed with the same hashing algorithm as previously described in chapter 4.4.7 (line 15). If newly created hash is stored in an array from the smart contract, a success message will be shown (line 20). Otherwise, an error message will be displayed (line 22).

```
1  handleFormSubmit = () => {
2  [...]
3      //get user input
4      let fname = this.state.fname;
5      let lname = this.state.lname;
6      let idNum = this.state.idNum;
7      let kycKey = this.state.kycKey;
8
9      //get array of hashes from smart contract
10     contract.methods
11       .getHashes()
12       .call()
13       .then(result => {
14         //hash the user data to compare the hash
15         let userHash = web3.utils.soliditySha3(
16           `${fname} ${lname} ${idNum} ${kycKey}`
17         );
18         //check if the user hash is in the array from smart
              contract
19         if (result.includes(userHash) === true) {
20           this.setState({ successMessage: "Your KycKey is valid!"
                 });
21         } else {
22           this.setState({ errorMessage: "Your KycKey is not valid!
                 " });
23         }
24       });
25 [...]
26   };
```

Listing 4.12: KYC key validation

Figure 4.8 shows the validation page from the user's perspective. In this example, the KYC key is valid. Since the information is strictly confidential, the input fields had to be blurred.

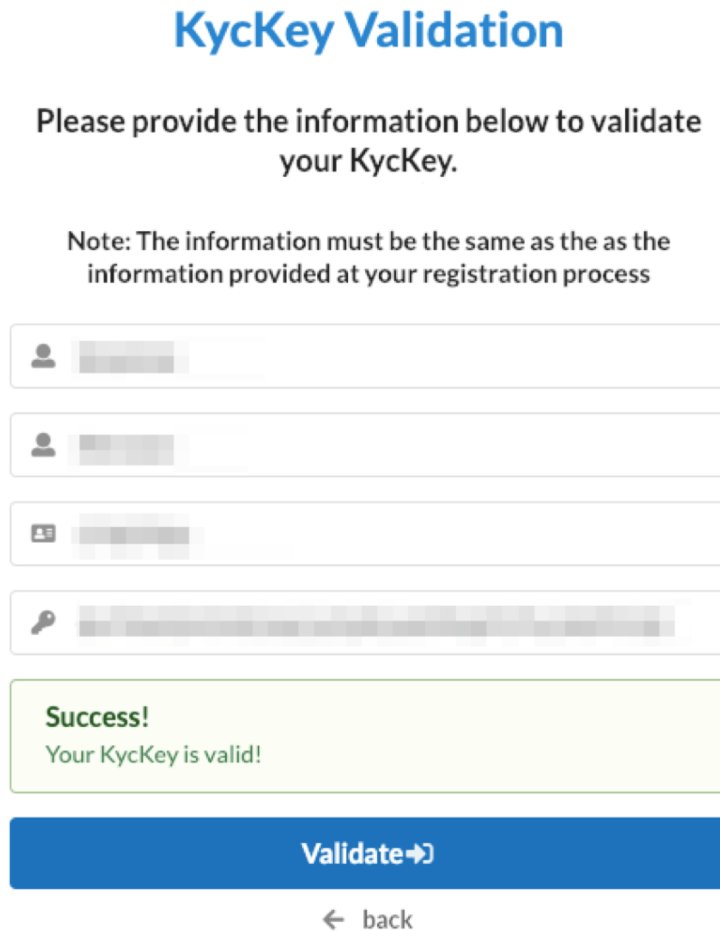

Figure 4.8: Validation of the KYC key

4.6 Admin

The landing page of an administrator after login is the *admin* page. It is important that he will always first be subscribed to the Pusher channel to be able to receive the notifications whether a user is requesting a video identification. The other feature on the admin page is the access to all user profiles. The administrator can see all users who have registered, also those who haven't completed the video identification yet. This gives a good overview of how many users have registered and which is going to call soon. Also, the administrator can access every profile of the users containing their personal information, pictures and the audio recording. Important to note here is that the administrator has no access to the user's KYC key since only the user is entitled to use his KYC key.

4.7 System Architecture

To get a better understanding what was explained in the previous sections, figure 4.9 illustrates the system architecture.

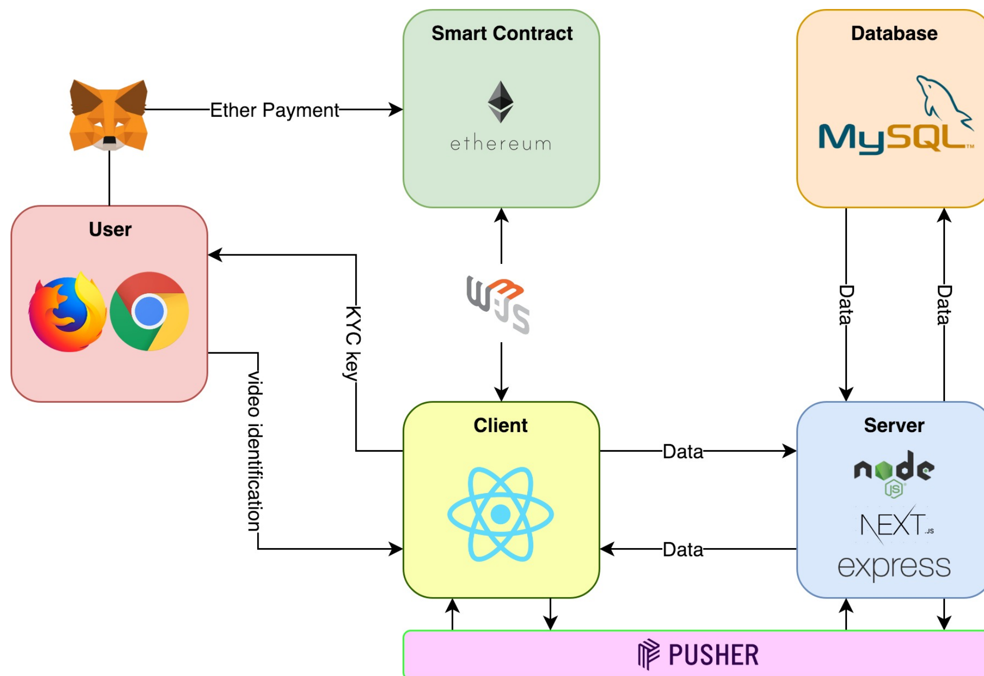

Figure 4.9: System Architecture

Chapter 5

The Smart Contract

This chapter explains how the smart contract was developed and what the intention was. Listing 5.1 illustrates the developed smart contract and will be referred to in the following sections.

```
pragma solidity ^0.4.17;

contract KYCVerification {
    address private kycAddress = 0x4399C3daed9b7cce56b7Edd4157FA3bDe3385d2A;
    bytes32[] array;

    //event listener for the Kyc Platform
    event KycListen(
        string kycKey,
        address platformAddress,
        address sender
    );

    //event listener for platforms requesting the verification of a kycKey
    event PlatformListen(
        bool confirmed
    );

    //send KycKey hash to the Kyc Platform to verify the KycKey
    function verify(string kycKey, address platformAddress) public payable {
        emit KycListen(kycKey, platformAddress, msg.sender);
        kycAddress.transfer(msg.value);
    }

    //send boolean answer to requesting address if kycKey is verified
    function answer(bool confirmed) public {
        require(msg.sender == kycAddress, "You cannot do this");
```

```
28          emit PlatformListen(confirmed);
29      }
30
31      //user payment for video identification
32      function payKYC() public payable {
33          kycAddress.transfer(msg.value);
34      }
35
36      //store hash of the user in bytes32 array
37      function storeHash(bytes32 newHash) public {
38          array.push(newHash);
39      }
40
41      //returns bytes32 array
42      function getHashes() public view returns(bytes32[]) {
43          return array;
44      }
45
46  }
```

Listing 5.1: Smart Contract handling interactions between application and the blockchain

5.1 Functionality

5.1.1 KYC Platform Address

First, the ethereum address of the KYC platform is set as a constant variable *kycAddress*. This is used to send the verification requests to the platform described in chapter 5.1.3.

5.1.2 Events

So called *events* can be used to show that something has occurred in the smart contract. JavaScript clients can listen for these events and react accordingly. The events are declared with the *event* keyword and emitted with the *emit* keyword [29]. For the verification requests, events had to be implemented. Between line 8 and 12 in listing 5.1 the event used by the kyc platform is defined. It contains variables such as the hashed KYC key, the platform's Ethereum address and the sender, which is the user's Ethereum address. The KYC platform listens for this event and once emitted from an external platform, the KYC platform receives the data contained in the event. To be able to listen for the events, an administrator always needs to be logged in.

The platform which requests the KYC key receives a response from the KYC platform in form of a boolean. For that, also an event is emitted and the platform listens for the boolean (line 15 to 17).

5.1.3 Verification Requests

The main function of the smart contract is to handle the verification requests for users who register at an external platform with their KYC key. The user will, once verified, have access to his KYC key through the profile page. With the KYC key the user will be able to register at a financial institution or a platform which requires identification. This section demonstrates a verification request with a trading platform developed by [15], which was successfully tested and is illustrated in figure 5.1.

First, the user delivers his KYC key to the trading platform. Since the KYC key is strictly confidential and the data transmitted through smart contracts is public, the trading platform needs to hash the KYC key with a *sha3* hashing algorithm before sending it to the KYC platform.

Secondly, the function *verify* is triggered by the user's ethereum address. Together with the hashed KYC key, the external platform's Ethereum address is also sent, in order to prove which user has registered at which platform (line 20). The *KycListen* event will then be emitted (line 21) and a defined ether value is transferred to the KYC platform (line 22).

Thirdly, the data will be sent to the KYC platform, which is listening to the *KycListen* event. Once received, the KYC platform checks the database for the KYC hash entry.

Fourthly, the function *answer* (line 26) will be triggered containing a boolean if the hash of the KYC key was found or not. It is required that the sender of this transaction is the KYC platform, otherwise an error message will be shown (line 27). In the end the *PlatformListen* event is emitted to ensure that the trading platform listens for the boolean.

Fifthly, the smart contract transmits the boolean to the trading platform.

Sixthly, the trading platform checks the boolean received and grants the user access to the platform if the boolean was true.

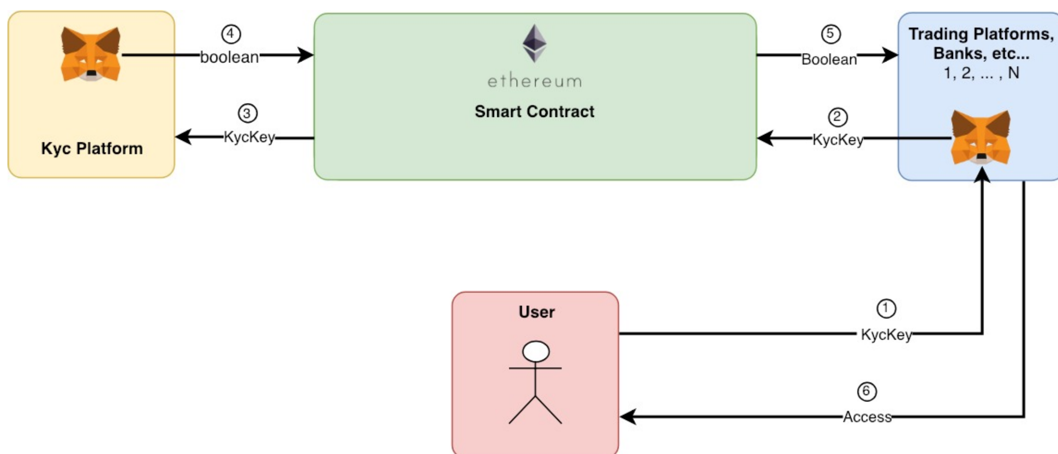

Figure 5.1: Verification Requests

Using the smart contract to handle these verification requests gives the opportunity to track the requests back. This is important to find out what went wrong if an unauthorized user would access the external platforms.

5.1.4 Storing of user approvals for identity verification

At the beginning of the thesis a requirement was stated that the user's approval to request his data has to be stored on the smart contract. This has become unnecessary with the current system, because it can be assumed, that the user's consent is conducted when he registers at an external platform with his KYC key. Instead of storing the approval on the smart contract, it is possible to prove that the user has agreed that his data is being requested by tracking back the verification requests made. This means that costs can be saved and the smart contract can be kept simple.

5.1.5 Payment

As described in chapter 4.4.5 the user has the possibility to pay an amount of ether to the KYC platform for the identification service. This transaction is handled by the smart contract function *payKYC* (line 32-34). This payable function transfers a specific amount of ether to the *kycAddress* (line 33).

5.1.6 Storage of Hash

In order to prove the user's identity at any time, the hash of the user data, previously described in chapter 4.4.7, will be stored on the smart contract in a bytes32 array (line 5).

The function *storeHash* takes the hash and pushes it into the array (line 38). For the validation page explained in chapter 4.5, the function *getHashes* was developed. By calling this function the array with all stored hashes will be returned (line 43).

This is necessary, should the KYC platform's database be out of order or in maintenance. In this case the verification requests would not receive any response from the KYC platform. Therefore, the KYC key validation, see chapter 4.5, was implemented to have a verifiable proof of the user's identity at any time and in any circumstance.

5.2 Technical Aspects

5.2.1 Compiling

To compile the smart contract in JavaScript, the solc library can be used [55].

```
1 const kycPath = path.resolve(__dirname, "contracts", "
    KYCVerification.sol");
2 const source = fs.readFileSync(kycPath, "utf8");
3 //returns the bytecode and the interface used to deploy the
    contract
4 module.exports = solc.compile(source, 1).contracts[":
    KYCVerification"];
```

Listing 5.2: Compiling a smart contract

On line 2 the path to the contract is encoded in utf-8 and provided to the *solc.compile()* function (line 5) to compile the contract. The *solc.compile()* function returns an object which contains the contract bytecode and interface which are then used for the testing and the deployment.

5.2.2 Deployment

To deploy the contract a *deploy.js* file was implemented. The smart contract will be deployed to the *Rinkeby Testnet*, which operates the same as the main Ethereum Network, except for the fact that ether does not have any value [47]. The smart contract is deployed from an Ethereum account and therefore needs to be authenticated. This is achieved with *HDWalletProvider*, which derives an account from a *mnemonic* (line 5) [10]. The *mnemonic* phrase contains twelve random words and serves as password, which grants access to an Ethereum account (line 7). Also, an *Infura* API key is required, which allows us to access the *Rinkeby Testnet* (line 9) [11]. Finally, the *provider* constant is accessed from the *Web3* instance (line 11). The contract is then deployed with the *interface* and the *bytecode* delivered from *compile.js* in listing 5.2.

```
1 const HDWalletProvider = require("truffle-hdwallet-provider");
2 const Web3 = require("web3");
3 const { interface, bytecode } = require("./compile");
4
5 const provider = new HDWalletProvider(
6   // mnemonic
7   "Your 12-word account seed phrase",
8   // Personal Infura API Key
9   "https://rinkeby.infura.io/v3/b652b579ac1f4420a12407a0a1836b44"
10 );
11 const web3 = new Web3(provider);
12
13 const deploy = async () => {
14   const accounts = await web3.eth.getAccounts();
15
16   console.log("Attempting to deploy from account", accounts[0]);
17
18   const result = await new web3.eth.Contract(JSON.parse(interface)
      )
```

```
19     .deploy({ data: bytecode })
20     .send({ gas: "5000000", from: accounts[0] });
21
22   //log the interface and contract address
23   //they are used in the contract.js file
24   console.log(interface);
25   //address of deployed contract
26   console.log("Contract deployed to", result.options.address);
27 };
28
29 deploy();
```

Listing 5.3: Smart Contract Deployment

5.2.3 Testing

Before deployment, it is necessary to test the implemented functions to ensure that they behave as expected. Once compiled, a test file was written with mocha, a simple JavaScript test framework [31]. To test invariants with assertion tests, the Node.js library *Assert* was imported (line 1) [36]. Secondly, ganache, a blockchain emulator, is required to make blockchain calls without deploying the smart contract to an actual Ethereum node (line 2) [34]. The third required import is the contract *interface* and the *bytecode*, which is provided after compiling the smart contract (line 7).

Before testing, the *BeforeEach()* function will run. This function describes what has to be done before the testing. On line 14, an Ethereum account has to be fetched to deploy the contract. On line 17 a new contract instance is created, which will be deployed to ganache on line 19 using the bytecode from *compile.js*.

After the *BeforeEach()* call, the contract is deployed to the blockchain emulator ganache, allowing us to write the test functions. The first function is to verify if the contract has been deployed (line 28 to 30). On line 31 the function *verify()* from the smart contract (listing 5.4 line 20) is tested. For this, a KYC key and a platform address is inserted to observe the behaviour of the function. Similarly, the next function checks if the *answer()* function returns a boolean to the receiver. To test the payable funtion *payKYC()*, an amount of ether is sent from an account (line 42 to 46). On line 48 the test is written to check whether a created hash is able to be stored on the smart contract. First, the hash is created with the function *web3.utils.soliditySha3()* and is served with four strings (line 49 to 54). Then, the method *storeHash()* is executed with the just created hash (line 55). Lastly, the functionality of getting the array containing all hashes from the smart contract is tested. Hence, the *getHashes()* method is called (line 58).

```
1 const assert = require("assert");
2 const ganache = require("ganache-cli");
3 const Web3 = require("web3");
4
5 const provider = ganache.provider();
6 const web3 = new Web3(provider);
```

```
7 const { interface, bytecode } = require("../compile");
8
9 let accounts;
10 let kycverify; //javascript representation of Contract
11
12 beforeEach(async () => {
13   //Get a list of all accounts
14   accounts = await web3.eth.getAccounts();
15   // Use one of those accounts to deploy the contract
16   // teaches web3 about what methods an User contract has
17   kycverify = await new web3.eth.Contract(JSON.parse(interface))
18     //tells web3 that we want to deploy a new copy of this
         contract
19     .deploy({ data: bytecode })
20     //Instructs web3 to send out a transaction that creates this
         contract
21     .send({ from: accounts[0], gas: "5000000" });
22
23   kycverify.setProvider(provider);
24 });
25
26 //Test all functions of the smart contract
27 describe("KYCVerification", () => {
28   it("deploys a contract", () => {
29     assert.ok(kycverify.options.address);
30   });
31   it("transfers a payable message ", async () => {
32     await kycverify.methods.verify(
33       "0xbE731D43973446dDDD5dbE2548047539f5C129f2",
34       "0x89B4837424cf559CC36112FD357B25Ba423B234B"
35     );
36   });
37   it("can send a boolean answer", async () => {
38     await kycverify.methods.answer(true);
39   });
40   it("can pay an either amount", async () => {
41     await kycverify.methods
42       .payKYC()
43       .send({
44         from: accounts[0],
45         value: web3.utils.toWei("0.1", "ether")
46       });
47   });
48   it("can store a sha3 hash", async () => {
49     let hash = web3.utils.soliditySha3(
50       "firstName",
51       "lastName",
52       "kycKey",
53       "idNumber"
54     );
```

```
55     await kycverify.methods.storeHash(hash);
56   });
57   it("can return an array of all stored hashes", async () => {
58     await kycverify.methods.getHashes();
59   });
60 });
```

Listing 5.4: Testing the Smart Contract

When we run the *KYCVerification.test.js* file, it will produce the following output illustrated in figure 5.2.

```
✓ deploys a contract
✓ transfers a payable message
✓ can send a boolean answer
✓ can pay an either amount
✓ can store a sha3 hash
✓ can return an array of all stored hashes


6 passing (794ms)
```

Figure 5.2: Mocha test result

5.2.4 Gas

When interacting with a deployed smart contract on the Ethereum network, the user must pay *gas* in *Ether* for the computation of every transaction he sends. When sending transactions to the Ethereum network, you can define the parameters *gasLimit* and *gasPrice*. *gasLimit* is defined as the maximum units of gas required for a transaction and depends on how much code is executed on the blockchain. The necessary amount of *gas* increases the complexity of the code, *e.g.* loops, and thus the *gasLimit* has to be specified to cover the computational resources. If it is too low, the transaction will fail. The other parameter, *gasPrice*, represents which amount the user is willing to pay per unit of *gas*. It indicates how quickly the transaction will be mined. Hence, by increasing the *gasPrice* your transaction will be mined quicker [33].

To prevent the transaction from failing, the *gasLimit* should be set rather too high than too low. In this application the *gasLimit* was set to 200'000 which is much more than each transaction costs but since the user gets the unused amount of gas for the transaction back, the user will never pay more than necessary.
In this system, users only interact with the smart contract once, namely for the Ether payment before the video identification process. The administrator of the system makes more transactions, but these can be executed in parallel to other functions in the system. Thus, the transactions are not time crucial and to keep the costs low, the *gasPrice* was set to 5'000'000'000 Wei.

Chapter 6

Evaluation

This chapter evaluates the system performance of how good the system performs in terms of costs, waiting time, security and scalability.

6.1 Costs

The user only has to make a one-time payment in Ether. Transferring 0.1 Ether to the KYC platform costs the user approximately 10.45 Swiss Francs. Since the user can enter a desired value or also choose not to pay, the effective costs cannot be determined.

On the other hand, the administrator has recurring costs which arise through answering to the verification requests and storing the hash on the smart contract. Since for both transactions it is not needed to send any funds along, the costs are reduced to a minimum. Answering to the platform which sent a verification request, it will cost the administrator 0.000115 Ether which is approximately 0.012 Swiss Francs. The costs to store the hash on the smart contract amount to 0.000244 Ether, which is about 0.03 Swiss Francs.

For processing the verification process of the trading platform however, the administrator gets a compensation of 0.01 Ether (1.03 Swiss Francs) [15].

If we sum up these outflows and inflows, we can see that they cancel each other out which means that the administrator has practically no costs or even has earned a small amount. In long terms, assuming that users register at more than one platform and that they pay at least a little amount for the video identification, the administrator will be compensated for the KYC processes provided.

6.2 Waiting Time

Another important factor to measure the performance of the platform is the waiting time. Ethereum transactions take some time until the transaction is mined. This can decrease the degree of user friendliness in the platform.

Fortunately, the users only have to wait once for a transaction to be processed, namely at the Ether payment. If the user decides to send a value to the platform his waiting time is approximately 20 seconds. The time can vary since the it depends on how busy the network is. However, the user also has some waiting time when accessing the video chat. Since the administrator is in charge of calling the users, it can occur that there are many users waiting in the pipeline to be called. In that case users would have a longer waiting time. For future work it could be considered to subscribe to multiple channels allowing many administrators to conduct the video identifications.

In the case that the administrator transmits transactions, the waiting time for him increases. For example, the storage of the hash takes about 35 seconds. By answering the verification requests additional 16 seconds could arise. This is not so bad for the administrator, since both transactions are sent in the administrator page allowing the administrator to keep on working while waiting for the transaction to be processed. A similar use case occurs during the video transmission with a user. The OCR scanning and MRZ validation take some time to perform. Luckily, the administrator has the possibility to trigger the OCR scan while identifying his counterpart because the scan takes up to 1 minute depending on how the image was cropped and the image quality. It has to be noted that the scanning time was significantly shorter when cropping the image generously. Still, there should be no character in the cropped image besides the MRZ code, otherwise the time will increase and the validation will fail.

6.3 Scalability

For future work, the scalability could be interesting in order to develop additional functions to the system. This section discusses, if the video chat, the smart contract or the database could limit the scalability of the system.

Since only a small amount of data is stored on the smart contract, the scalability is only partially affected. Only by a growing amount of hashes stored in the smart contract array, costs would increase. Since hashes should be dropped from the array, when a user has to re-register, sooner or later the costs will compensate each other.
Databases, such as MySQL can process a large amount of data and therefore are not a limitation for the system's scalabality.

However, the video chat, could decrease in performance when there is a large quantity of users using the system simultaneously. This was tested with up to five users from five different browsers accessing the video chat channel on the same computer. The user interface turned out to be laggy. Since, all users were logged in on the same computer, it could be that the computer caused the system to slow down, since all browsers were accessing the video transmission. This should be fully tested in the future with a deployment of the system and measure the performance of the video chat from different computers.

6.4 Security

For the system developed in this system, it is crucial to be aware of the security concerns, since there is sensitive data stored in the database. The platform is publicly accessible and there is need of securing the system.

First, the *CSRF* attacks are considered. As described in chapter 3.1, these attacks aim to trick the browser to send unwanted HTTP requests. Since the performed tests were successful, there is no need to take more action on this topic.
A second concern is the SQL Injection, which is a system vulnerability concerning the SQL database. This allows attackers to send malicious SQL commands to the database and thus allowing the attackers to access or delete data from the database. This can be prevented by escaping the user input before enter them to the query [57]. In this system, there are many possibilities for an attacker to send an input to the server through a form. To close this security gap, the *sqlstring* package for npm was used, which *escapes* all the input [17].

The database of the developed platform stores some strictly confidential data, such as the KYC key. It is crucial that the does not fall into the wrong hands since it allows users to register anonymously at external platforms. As described in chapter 4.1.1 the user's password is hashed preventing external attackers from unauthorized access. Also, a strong password for the authorization is required for the user making it harder to be hacked (chapter 4.2.4). These aspects also hinder the administrator from logging into user profiles since he has almost full access to the database. Another consideration has to be done, namely the question how safe is a user's KYC key. Since the administrator has no access to the KYC keys of the users, attackers could only access the user's profile, when they somehow get the password and the user name of a user.

Lastly, the security aspects of smart contract has to considered, since it could also be a vulnerability. For future work, security inspections have to be done by specialists to rule out any security breach before deploying the smart contract onto the main Ethereum network.

Chapter 7

Summary and Conclusions

The goal of this thesis was to develop an open source and smart contract-based know your customer (KYC) platform which provides video identification for users and enables them to register themselves at a connected platform completely anonymous with their proof of verification (*KYC key*).

Firstly, a secure registration platform was built to enable users to register themselves in a secure and user-friendly way. A database was connected to the platform to store and read the user's data. Since the platform requires authorization from users and contains strictly confidential data, the platform had to be secured with so called JSON Web tokens (JWT). Storing these tokens in the cookies makes the platform secure against CSRF attacks. Also, with help of these tokens users cannot access pages, for which they are not entitled. To prevent users spamming the platform with fake email addresses, email-based verification was implemented to get proof that the user knows his email address and is in possesion of the password.

In a second step, the implementation of the video identification process was approached. To ensure that the video identification complies to the regulations published by FINMA, the requirements had to be identified. All requirements were implemented using a number of external libraries as well as WebRTC to guarantee a interrupt-less and secure onboarding process. The goal of the video identification process is to provide a KYC service to the user, who has to pay for this service. After successful completion the user receives a proof of verification from the platform. This proof of verification, the *KYC key*, is a unique hash, which allows the user to register at external platforms anonymously.

Furthermore, a smart contract was developed to handle verification requests from external platforms, such as the trading platform of [15]. Since the trading platform allows registrations with only the *KYC key*, they need a proof that the user holding the key really has identified himself. In this case the trading platform can request a verification of the KYC key by hashing it and sending it to the KYC platform, which verifies the hashed key and sends back a message saying if the KYC key is valid. Another functionality of the smart contract serves the purpose of giving the users the possibility of validating their KYC key at any time, even if the platforms database is out of order. By hashing the first name, last name, identity card number and the KYC key of the user and storing it

in the smart contract at the end of the identification process, the user is able to enter the mentioned data in a page of the platform to validate his KYC key. The last smart contract functionality handles the user payment. Since it is assumed, that all users are in possession of an Ethereum account, the payment is transacted with Ether.

The great advantage of video identifications is that on-site onboarding at financial institutions becomes unnecessary. This limits the amount of KYC processes to a minimum for the institution and for their customer. Also, the need of physical paper documents is no longer needed since the customer can sign the contracts digitally with the digital identity created in the identification process. Another key benefit is the cost reduction for the financial institution and their customer. On the one hand users only have to identify themselves once until they receive new identification documents. This means that users only have to pay once. The financial institution also only has to onboard the customer once and receives a small compensation for the KYC process for each verification request made from an external platform.

Since the implementation of the verification request function in the smart contract is strongly dependent on external platforms, the development of this thesis was easier thanks to the parallel development of the trading platform of [15] enabling a collaboration between the platforms. Without this collaboration, the testing, and the identification of the requirements, would have been very difficult.

In summary, the prototype developed om this thesis offers a new approach on how to provide KYC processes more efficient for financial institutions and platforms which require an identification. All the requirements defined at the beginning of the thesis as well as the requirements published by FINMA are met. The developed system performs reliably in terms of its functions.

Chapter 8

Future Work

Suggestions for future work are described in the following paragraphs.

In the case that users receive new identification documents, for example when the old ones have expired or their name changes after marriage, the user will have to re-register in the system and go through a new video identification process. The user will receive a new KYC key after re-registering and therefore, it is important that not only the KYC platform handles re-registrations, but also [15] has to be notified that the current KYC key of this user is invalid. Hence, the user has to be blocked until he provides the new KYC key to the platform.
When an anonymous user in [15] has been reported in case of fraudulent behaviour, the user data is required for further investigation. In this case, the smart contract could be modified to handle user data requests to the KYC platform, in which the user's data would be delivered to the trading platform.

The database of the KYC platform contains strictly confidential data, such as the KYC key as well as the hash of the KYC key. Therefore, the database should be analyzed and secured by specialists to ensure that no unentitled users have access. This also requires a detailed inspection of the SQL statements interacting with the database.

As seen in chapter 2, most of the companies providing KYC solutions are mobile based. Developing a mobile application of the KYC platform could enhance the user experience since mobile apps are accessible in many places at any time. This complies with our modern lifestyle. Furthermore, the performance could be improved with for example, built in MRZ scanning solutions.

Bibliography

[1] Aboukhadijeh, F.: Simple-peer; https://github.com/feross/simple-peer, November 26, 2018.

[2] Anti-Money Laundering Ordinance (AMLO-FINMA): Verordnung der Eidgenössischen Finanzmarktaufsicht über die Bekämpfung von Geldwäscherei und Terrorismusfinanzierung im Finanzsektor; https://www.admin.ch/opc/de/classified-compilation/20143112/index.html, January 24, 2019.

[3] Axios: Promise based HTTP client for the browser and node.js; https://github.com/axios/axios, November 26, 2018.

[4] Blockchain Expo: Selfkey KYC-Chain; https://blockchain-expo.com/global/partners/selfkey-kyc-chain/, November 27, 2018.

[5] BlockchainHub: Blockchain Glossary; https://blockchainhub.net/blockchain-glossary/, November 28, 2018.

[6] Blockgeeks: What is Blockchain Technology? A Step-by-Step Guide For Beginners; https://blockgeeks.com/guides/what-is-blockchain-technology/, January 29, 2019.

[7] Campbell, N.: Bcrypt; https://github.com/kelektiv/node.bcrypt.js, November 26, 2018.

[8] CB Financial Services AG: CBFSIdent; https://www.c-b-f-s.com/services/cbfsident/, November 27, 2018.

[9] Cheminfo: mrz package for npm; https://github.com/cheminfo-js/mrz, January 29, 2019.

[10] Consesus Systems: Truffle HDWallet Provider; https://github.com/trufflesuite/truffle-hdwallet-provider, January 28, 2019.

[11] Consensys: Infura; https://infura.io/, January 28, 2019.

[12] De Feniks, R., Reverelli, R.: Tradle: KYC on blockchain; http://www.digitalinsuranceagenda.com/108/tradle-kyc-on-blockchain/, January 24, 2019.

[13] Deri, A.: laravel-react-webrtc-video-chat; https://github.com/AfikDeri/laravel-react-webrtc-video-chat, January 30, 2019.

[14] FINMA: Circular "Video and online identification"; https://www.finma.ch/en/~/media/finma/dokumente/rundschreiben-archiv/2016/rs-16-07/finma-rs-2016-07.pdf?la=en, November 7, 2018.

[15] Gabay, A.: Distributed and Decentralised Proof of Ownership with Blockchains; https://www.csg.uzh.ch/theses/theses.html?thesisid=252, 2019.

[16] Garage inside Garage Blog, Secure JWT Authentication Against both XSS and XSRF (VUE.js + DJANGO REST). https://blog.garageinsidegarage.com/secure-jwt-authentication-against-both-xss-and-xsrf-vue-js-django-rest/, January 29, 2019.

[17] Geisendörfer, F.: sqlstring package for npm; https://github.com/mysqljs/sqlstring, January 30, 2019.

[18] Hody, P.: Raiffeisen: So cool kann Schweizer Fintech sein; https://www.finews.ch/news/banken/24642-fintech-cb-financial-services-raiffeisen-marcel-komminoth-roland-ruettimann, January 24, 2019.

[19] Holowaychuk, TJ., Wilson, D.C.: Cookie-Parser; https://github.com/expressjs/cookie-parser, November 26, 2018.

[20] Holowaychuk, TJ., Shtylman, R., Wilson, D.G.: Express.js; https://github.com/expressjs/express, November 26, 2018.

[21] IDNow: API; https://www.idnow.io/development/api-documentation/, November 27, 2018.

[22] IDNow: Auto-identification; https://www.idnow.io/products/idnow-autoident/, November 27, 2018.

[23] IDNow: Regulation; https://www.idnow.io/regulation/identification-kyc/, November 27, 2018.

[24] IDNow: Video-identification; https://www.idnow.io/products/video-verification/, November 27, 2018.

[25] JWT: Introduction to JSON Web Tokens; https://jwt.io/introduction/, November 26, 2018.

[26] Kienböck, R.: Blockchain: Was sind eigentlich Smart Contracts?; https://futurezone.at/digital-life/blockchain-was-sind-eigentlich-smart-contracts/293.031.908, November 28, 2018.

[27] KYC-Chain: KYC-Chain; https://kyc-chain.com/, November 27, 2018.

[28] Lielacher, A.: Deloitte's RegTech Offering: Blockchain-Powered KYC-as-a-Service Solution; https://bitcoinmagazine.com/articles/deloittes-regtech-offering-blockchain-powered-kyc-service-solution/, January 24, 2019.

[29] Marx, S.: Logging and Watching Solidity Events; https://programtheblockchain.com/posts/2018/01/24/logging-and-watching-solidity-events/, January 28, 2019.

[30] MetaMask: MetaMask; https://metamask.io/, January 28, 2019.

[31] Mocha.js: Mocha.js; https://mochajs.org/, January, 2019.

[32] Moyano, J. P., Ross, O.: KYC optimization using distributed ledger technology; Business & Information Systems Engineering, 59(6), 411-423, 2017.

[33] MyEtherWallet: What is Gas?; https://kb.myetherwallet.com/gas/what-is-gas-ethereum.html, January 28, 2019.

[34] Nethereum: Ganache CLI; https://nethereum.readthedocs.io/en/latest/ethereum-and-clients/ganache-cli/, January 29, 2019.

[35] Next.js: Next.js; https://nextjs.org/, November 26, 2018.

[36] Node.js: Assert; https://nodejs.org/api/assert.html, January 28, 2019.

[37] Node.js: Crypto; https://nodejs.org/api/crypto.html, November 26, 2018.

[38] Nodemailer: Nodemailer; https://nodemailer.com, November 26, 2018.

[39] NorthRow: KYC Checks for Customer/Client Onboarding & Monitoring; https://www.northrow.com/kyc-checks/, November 26, 2018.

[40] Ong, J., Wilson, D.C.: Node.js body parsing middleware; https://github.com/expressjs/body-parser, November 26, 2018.

[41] Project Naptha: tesseract.js; https://github.com/naptha/tesseract.js, January 29, 2019.

[42] Pusher: Pusher; https://pusher.com/, November 26, 2018.

[43] Pusher: Client Events; https://pusher.com/docs/client_api_guide/client_events, November 29, 2018.

[44] Pusher: Documentation; https://pusher.com/docs, November 26, 2018.

[45] React.js: JSX; https://reactjs.org/docs/introducing-jsx.html, November 26, 2018.

[46] React.js: React.js; https://reactjs.org/, November 26, 2018.

[47] Rinkeby Testnet: Rinkeby Testnet; https://www.rinkeby.io/, January 28, 2019.

[48] Semantic UI React: Semantic UI React; https://react.semantic-ui.com/, November 26, 2018.

[49] Sironi, G.: Cross-Site-Request-Forgery explained; https://dzone.com/articles/cross-site-request-forgery, January 29, 2019.

[50] Shufti Pro: Shufti Pro; https://shuftipro.com/, November 27, 2018.

[51] Shufti Pro: Identity-verification; https://shuftipro.com/identity-verification/, November 27, 2018.

[52] Shufti Pro: Integration; https://shuftipro.com/integration/, November 27, 2018.

[53] Shufti Pro: Modes of verification; https://shuftipro.com/modes-of-verification, January 25, 2019.

[54] Solidity: Documentation; https://solidity.readthedocs.io/en/v0.4.24/, November 27, 2018.

[55] Solc: Solc compiler for npm; https://github.com/ethereum/solc-js, January 24, 2019.

[56] SweetAlert2: SweetAlert2; https://sweetalert2.github.io/, January 28, 2019.

[57] Veracode: What is SQL Injection?. https://www.veracode.com/security/sql-injection, January 30, 2019.

[58] W3Schools: Cookies; https://www.w3schools.com/js/js_cookies.asp, Januray 30, 2019.

[59] W3Schools: Node.js MySQL; https://www.w3schools.com/nodejs/nodejs_mysql.asp, November 26, 2018.

[60] Web3.js: Web3.js; https://web3js.readthedocs.io/en/1.0/, November 26, 2018.

[61] WebRTC: WebRTC; https://webrtc.org/, November 27, 2018.

[62] 3CX: WebRTC FAQ & Info; https://www.3cx.com/webrtc/, November 27, 2018.

[63] Yeo, G.: Otplib; https://github.com/yeojz/otplib, January 28, 2019.

Abbreviations

AI	Artificial Intelligence
AML	Anti-Money Laundering
API	Application Programming Interface
CSRF	Cross-Site-Request-Forgery
DApp	Decentralized Application
DLT	Distributed Ledger Technology
FINMA	Eidgenössische Finanzmarktaufsicht Schweiz
HTTP	Hypertext Transfer Protocol
ICO	Initial Coin Offering
IPC	Inter-process communication
JSON	JavaScript Object Notation
JWT	JSON Web Token
KYC	Know Your Customer
MRZ	Machine Readable Zone
P2P	Peer-to-Peer
PoC	Proof of Concept
RTC	Real-Time Communications
SaaS	Software as a Service
SC	Smart Contracts
OCR	Optical Character Recognition
OTP	One Time Password
UI	User Interface

Glossary

API : Application programming interfaces provide a standardized way to exchange information between individual programs and an application

Peer to Peer : Direct communication between two nodes through a network.

List of Figures

4.1 Email confirmation link 18
4.2 Video identification process 20
4.3 Registration Form 21
4.4 Success Message 22
4.5 Enter beneficial owners of the assets 23
4.6 Terms and Conditions 24
4.7 Administrator page 28
4.8 Validation of the KYC key 34
4.9 System Architecture 35

5.1 Verification Requests 39
5.2 Mocha test result 44

B.1 Request a re-registration button 71

List of Tables

2.1 Comparison of requirements 8

3.1 Requirements for video identification from FINMA Circular 2016 [14] . . . 10

3.2 Requirements for video identification from AMLO-FINMA [2] 11

Listings

4.1 Signing a JWT . . . 16
4.2 Verifying tokens . . . 16
4.3 Sending email verification links . . . 17
4.4 Setting cookies into the Browser . . . 20
4.5 Client-side server request . . . 21
4.6 Ether payment . . . 25
4.7 Bind client event to the Pusher channel . . . 26
4.8 Establish P2P connection . . . 26
4.9 Identity Card Validation . . . 28
4.10 OTP verification . . . 30
4.11 Create the KYC key . . . 31
4.12 KYC key validation . . . 33
5.1 Smart Contract handling interactions between application and the blockchain 37
5.2 Compiling a smart contract . . . 41
5.3 Smart Contract Deployment . . . 41
5.4 Testing the Smart Contract . . . 42
B.1 Create a button . . . 71
B.2 Client-side request to delete a user's data . . . 71
B.3 Delete a user's data from the database . . . 72

Appendix A

Installation Guidelines

This installation guideline is based on MacOSX and may differ from Windows installation

Guideline without Docker

1. **Clone Github Repository**
   (a) Download Visual Studio Code (or similar): https://code.visualstudio.com/download
   (b) Create Github account if not already existant
   (c) Clone Github repo in VS Code terminal: git clone https://github.com/sealle/BA-S.E.A
   (d) If required, download the git command line developer tools (Mac)
   (e) Open the git repository from Folder in Vscode (File -> Open): Users/"yourUsername"/BA-S.E.A
   (f) Install solidity and MySql extension in Vscode
2. **Download and Set up MySQL server**
   (a) Download *MySQL Community Server*: https://dev.mysql.com/downloads/mysql/
   (b) Download *MySQL Workbench* (optional): https://dev.mysql.com/downloads/workbench/
   (c) Open MySQL Workbench and create a new connection
   (d) In VS Code, press the + button in the MySQL extension and enter *host, user, password and port* (from MySQL Workbench connection)
   (e) If there is an Error message regarding your *localhost* database, execute the following query in MySQL Workbench: 'ALTER USER *"your_user"*@*"your_host"* IDENTIFIED WITH mysql_native_password BY *"your_password"*;'
   (f) Execute the SQL queries stored in the *users.sql* file in MySQL Workbench
   (g) In *db.config.js*, insert the your credentials *host, user, password and database*

(h) Make sure your MySQL server is running every time your start your machine

3. **Install MetaMask**

   (a) Add the MetaMask extension in your browser

   (b) In MetaMask, create new account or import existing account with seed phrase

   (c) Allow MetaMask in Incognito Mode (Google Chrome): Settings -> Extensions -> Metamask -> Allow in Incognito

   (d) Open `https://infura.io` and sign up. Then you can create your own API key for your desired network. You then need to copy and paste your MetaMask mneumonic and the infura API key in *deploy.js* on lines 6 and 7.

   (e) Insert the address of your MetaMask account into the *KYCVerification.sol* file as *kycAddress* variable

4. **Pusher API**

   (a) Sign up with Pusher: `https://dashboard.pusher.com/accounts/sign_up`

   (b) Create new App and select *React* and *Node js* as technlogies and *eu* as cluster

   (c) Paste your App Key in *VideoChatUser.js* on line 5 and the cluster on line 107

   (d) Paste your App Key in *AdminConsole.js* on line 5 and the cluster on line 150

   (e) Insert your *appId, key, secret, cluster and encrypted* in *server.js* on line 40-46

5. **Nodemailer**

   (a) In *server.js* on lines 32,34,35, enter the *Service* (*e.g.* Gmail), *Admin email address* and the *password*

   (b) In *server.js* on line 26, enter an EMAIL_Secret for nodemailer

   (c) On lines 169, 316, 535, 566, 611, 861 enter your nodemailer email account in the *server.js* file

6. **JWT**

   (a) In *server.js* on line 24 enter a secret as jwt secret

7. **Getting Started**

   (a) If not already installed, install *Node.js*: `https://nodejs.org/en/download`

   (b) Run *npm install* in the VS Code terminal (Attention: *simple-Peer* version 9.0.0 required (higher will not work))

   (c) Enter *cd ethereum* in the VS Code terminal and run *node compile.js*

   (d) Then, still in the ethereum directory, run *node deploy.js*. The results will be logged in the console. Copy the *contract address* and the *contract ABI* and paste them into the *contract.js file.*

   (e) Choose the folder where you want to store the images, documents and audio recordings of the user in the *server.js* file on the lines *169, 172, 306, 309, 312, 315, 893 and 972.*

(f) Run *npm start* to launch the application. It will be running on *localhost:3000* in your browser.

(g) To finally get started, register yourself as a person with the user name *Admin*. **After** confirming your email address, execute the following query in MySQL Workbench: 'UPDATE users SET privileges=*"admin"*, isRegistered=*"yes"* WHERE username=*"Admin"*;' (You may need to disable safe mode: Preferences -> SQL Editor -> Other)

Note: To test the video chat functions, the admin or the user must be in an incognito window

Appendix B

Modification Scenario

As describe in chapter 8, users with new identity documents must re-identify themselves. This chapter demonstrates how the functionality of deleting the user's data can be added to the platform.

First, a button is created in *render()* between the lines 832 and 833 in the *UserData.js* file. This button is used to trigger the re-registration process. In listing B.1 the button is created in a container. The button has a onClick event *this.reRegister*, which triggers *reRegister()* (line 2). Figure B.1 shows how the button presents itself on the profile page.

```
1 <Container style={{ display: "inline-block", textAlign: "center"
    }}>
2    <Button primary onClick={this.reRegister}>
3      Request a re-registration
4    </Button>
5 </Container>
```

Listing B.1: Create a button

Figure B.1: Request a re-registration button

As mentioned, the button triggers the function *reRegister* (Listing B.2). To delete the user's data, the function sends a request to the server. This is done with an *axios.post()* call to the defined endpoint */reRegister*. The *window.location.origin* statement is needed to ensure that the endpoint is attached to the current URL. The server will process the request and send back a response to the client, where the response is verified. As we see on line 4, the user will receive a success message and will redirect the user to the register page, if the server request was successful (line 5). This function can be added in the *UserData.js* file.

```
1 reRegister = async () => {
```

```
2     let response = await axios.post(window.location.origin + "/
         reRegister");
3     if(response.data.success) {
4         swal("Success", "You will be redirected to the register
             page", "success");
5         Router.push("/register");
6     }
7 }
```

Listing B.2: Client-side request to delete a user's data

The goal of the server-side function (Listing B.3) is to delete the user's data in the database, so that the user can register himself again. In the *server.js* file a new *server.post()* call is created including the endpoint */reRegister* along with a function containing an input (*req*) and an output (*res*) (line 2). Since the data of a specific user must be deleted, the user name has to be identified. The user name is in an encoded token stored in the cookie, which is placed in the browser. With *req.cookies* you can get the *x-access-token* from the browser (line 3). The token is then decoded to get the user name (line 4 and 5). The SQL query is visible on line 6 to 10. Since there are three SQL tables, the data stored in all tables has to be deleted. The first query joins the *users* and the *ethAddresses* tables with a left join on the *kycHash* and deletes all rows containing the user name (line 7). The second query is to delete all beneficial owners from table with the same name (line 8).

To call the database, *database.connection.query()* is executed with the SQL query and also a callback function, which handles an error (*err*) and produces an output (*result*) (line 11). The function throws an error, if an error is detected (lines 12 and 13). Otherwise, a JSON object is sent back to the client containing the success variable set to *true*.

```
1 //deletes the database entries of the user
2 server.post("/reRegister", urlEncodedParser, function(req, res) {
3     let cookie = req.cookies["x-access-token"];
4     let decoded = jwtDecode(cookie);
5     let currentUser = decoded.username;
6     let deleteUser = SqlString.format(
7     "DELETE users,ethAddresses FROM users LEFT JOIN ethAddresses
         ON users.kycHash = ethAddresses.kycHash WHERE username=?;
         DELETE FROM beneficialOwners WHERE username=?",
8     [currentUser, currentUser]
9     );
10    database.connection.query(deleteUser, function(err, result) {
11        if (err) {
12            throw err;
13        } else {
14            res.status(200).json({
15                success: true
16            });
17        }
18    });
19 });
```

Listing B.3: Delete a user's data from the database

Appendix C

Contents of the CD

1. **Report**: The written report with all its source images and files and the final presentation
2. **Related Work**: Related work papers in PDF format
3. **Application**: The complete code implemented during this thesis