SCSE23-0200 : ChainTalk, A Decentralised Messaging Application

Final Report

Submitted by: Poon Yan Xin Melise

U2022504B

Date: 25 March 2024

Project Supervisor: A/P W. K. Ng

Examiner: A/P Goh Wooi Boon

School of Computer Science and Engineering

Academic Year 2023-2024

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering in Computer Science of the Nanyang Technology University (Singapore)

ABSTRACT

Blockchain technology have revolutionised various industries by providing decentralised, transparent and secure solutions. In this paper, we will document the developmental project, “ChainTalk”, an innovative communication application built on the blockchain.

ChainTalk aims to redefine traditional communication methods by leveraging the immutability and decentralisation of blockchain technology to ensure privacy, security and transparency in conversations.

This project encompassed a comprehensive design and implementation of both front-end and back-end components. The application architecture follows an innovative layered approach, including a robust front-end interface for user interaction and a back-end system powered by smart contracts deployed on the blockchain. These smart contracts govern the core functionalities of ChainTalk, including decentralised storage and access control.

To enhance user identity management, ChainTalk implements cutting-edge concepts such as ERC-735 standard to form an identity management system and Soulbound tokens for tamper-proof user identities.

Throughout the development process, design patterns and best practices in smart contract development were adhered to, to ensure security, efficiency, and maintainability. The project also undergoes testing to mitigate potential vulnerabilities and ensure a robust and reliable communication platform.

In summary, ChainTalk represents a pioneering effort in utilising blockchain technology to redefine communication paradigms. By providing a decentralised and secure platform for communication, ChainTalk aims to empower users with greater control over their data and privacy in the increasingly interconnected world.

ACKNOWLEDGEMENTS

The ChainTalk application is developed in fulfilment of the Degree of Bachelor of Engineering in Computer Science in Nanyang Technological University as a Final Year Project (FYP).

I would like to take this opportunity to thank my FYP Supervisor, Dr Ng for his guidance and feedbacks provided throughout this development process. The opportunity and feedbacks provided enabled me to explore the different new improvements and innovations made by the Blockchain Community and how I can use these technologies in my project.

I would also like to thank the Ethereum Stack Exchange Community that provided me with solutions throughout the development process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 : INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

In the digital era, mobile messenger applications have become an essential tool for communication. These applications allow users to exchange real time messages via the internet, enabling people from all walks of life to connect on a digital scale. With the widespread use of these applications, some of the industry giants, such as “WhatsApp” messenger boast a monthly user base of 2 billion and is one of the most popular mobile social media applications worldwide [1].

However, this also suggests that with more users on the digital frontier, individuals face a myriad of challenges related to the security and privacy of their personal data. The unauthorised sale of personal information and the lack of control over one’s own data have become a prevalent issue. Traditional, centralised entities are susceptible to data breaches and cyber-attacks, leading to compromised user information. There is also a growing concern related to the impermanence of digital communications [2] and potential for manipulation of data, particularly in formal discussions.

The motivation behind this project stems from the limitations of existing messaging applications and the potential of blockchain to address these challenges. The decentralised, immutable, and distributed nature of blockchain, together with its cryptographic principles, offers a promising foundation for building a secure, private, and user-centric messaging platform. By exploring this intersection of blockchain and messaging, the projects aim to redefine the standards of secure communication in the digital era.

1.2 PROBLEM STATEMENT

As our digital footprint continues to grow exponentially, users of centralised platforms often lack control of their own data, which can easily be monetised without their consent. In the recent years, privacy rights violations by several social media sites have increased worldwide, and users are increasingly concerned over the issue of data ownership. A notable incident includes the Cambridge Analytica incident where Facebook exposed data of up to 87 million users [3] .

A survey conducted by Pew Research Centre have stated that 80% of social media users were concerned about businesses accessing the data they share on social media platforms, with 59% agreeing that it would not be difficult to give up these sites [4]. With a lack of transparency in how messages are being stored and processed, users often have limited control over their data, resulting in the increase in the demand for tangible, decentralised alternatives to secure user data privacy. The decentralisation property of the blockchain remains a solution to this problem. It ensures that users now have control over their properties and do not have to rely on third-party to maintain and manage their assets.

Traditional messaging applications often rely on centralised servers and lack precautions and security mechanisms to safeguard user data, causing these platforms to be vulnerable to data breaches and unauthorised access. To put things into perspective, Meta itself, saw over 487 million user records such as user mobile numbers being put up for sale on a well-known hacking community forum in November 2022 [5]. It has been speculated that the threat actor most likely obtained these user credentials through harvesting information at scale, a technique also known as scraping. In light of these vulnerabilities, adoption of Blockchain technology's cryptographic features and authentication mechanisms promises enhanced security, mitigating data breach and reducing unauthorised access.

In addition to the challenges posed by data breaches, there is a need for digital permanence, which is to maintain data integrity and accessibility in the ever-evolving landscape of information storage [6]. Particularly in formal discussions such as business discussions among organisations or legal firms, chat messages need to be preserved and remain forever in the system. The immutability and distributed nature of the Blockchain ensures that data remains permanent and can be retrieved seamlessly, promising complete transparency. Furthermore, neither party in the agreement can make changes without the other party's knowledge. This ensures that data exchanged cannot be modified, deleted, or manipulated by either party, thereby providing a high degree of trust.

1.3 OBJECTIVE

The objective of this project is to create a secure, transparent, and decentralised messaging application, “ChainTalk”, that enables users to engage in instant messaging. With the integration of blockchain technology into the realm of digital communication, users will be able to create an account, add friends and exchange real-time messages with each other, with messages securely encrypted and only accessible by its intended recipient. With no intermediaries involved, ChainTalk will leverage the Ethereum blockchain for on-chain storage and InterPlanetary File System (IPFS) for off-chain storage, forming a layered architecture. To ensure the efficiency of message exchange, only the IPFS Content Identifiers (CIDs) of messages and its relevant metadata is stored on-chain. Messages will be stored in full in IPFS, leveraging its distributed file storage protocol. Furthermore, the usage ERC-735 standards and Soulbound tokens creates a digital identity for users, enhancing user autonomy in identity and minimises the risk of security information disclosure. ChainTalk’s unique architecture ensures tamper-resistant records and user data ownership, providing users a decentralised, secure, and efficient ecosystem to exchange messages in the digital communication sector.

1.4 SCOPE

The ChainTalk application encompasses of four integral components, each playing a crucial role in delivering a secure and decentralised user experience.

Firstly, the frontend interface is designed for user-friendly interactions, enabling users to create an account, add friends and send messages with each other. The frontend will be a web interface developed using ReactJS.

Secondly, the application backend will be built using smart contracts and to be deployed onto the Ethereum Blockchain. These smart contracts govern the core functionalities of the application, facilitating secure message sending, storage and retrieval. User registration, authentication, and user identity management are also some of the functionalities of the smart contracts.

Thirdly, IPFS nodes are strategically integrated for decentralised storage of message content exchanged among users.

Lastly, the Ethereum blockchain serves as the underlying infrastructure, providing the foundation for secure and transparent transactions.

The scope of this project is to develop a secure and decentralised messaging platform, using blockchain technology, empowering users with unprecedented control over their personal data and communication.

ChainTalk allows users to send messages with verified users that has been added as their friends. Real-time message synchronisation will also be implemented. Ethereum wallet addresses will be used as user identities to implement user authentication. Users can be further verified using their soulbound tokens. The smart contracts will also be deployed to an Ethereum Testnet, Sepolia for development testing.

The user interface of the application will be designed with intuitiveness in mind, accommodating users with seamless exchange of messages

Additionally, the project will include comprehensive documentation, detailing the technical aspects, functional requirements, system architecture, testing and user guides for seamless utilisation.

1.5 PROJECT SCHEDULE

This final year project will be presented to Nanyang Technological University in partial fulfilment of the requirements of the bachelor's degree of Engineering in Computer Science.

The project will be developed in 5 stages:

1. Project Planning & Analysis
2. System Design
3. Implementation
4. Testing
5. Project Deliverables & Evaluation

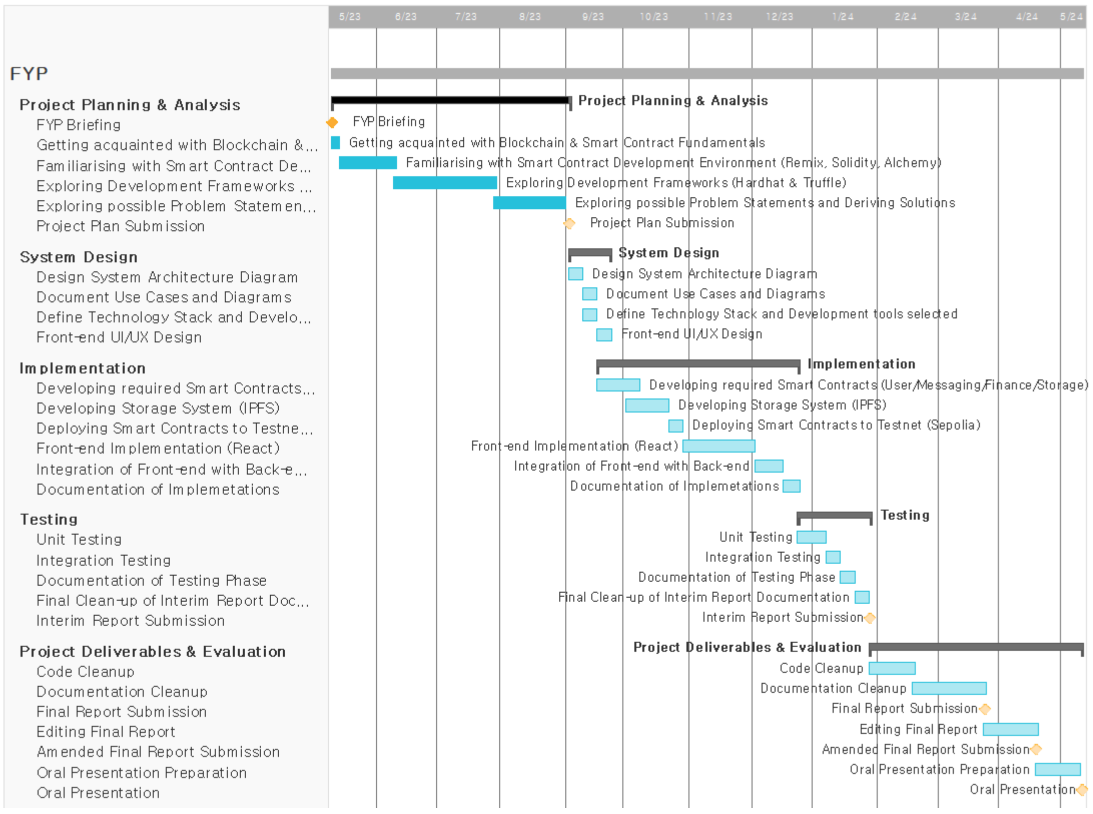


Figure 1: Project Schedule Gantt Chart

No.	Task	Start Date	End Date
Project Planning & Analysis			
1	FYP Briefing	10 May 2023	10 May 2023
2	Getting acquainted with Blockchain & Smart Contract Fundamentals	10 May 2023	15 May 2023
3	Familiarising with Smart Contract Development Environment (Remix, Solidity, Alchemy)	15 May 2023	12 Jun 2023
4	Exploring Development Frameworks (Hardhat & Truffle)	12 Jun 2023	31 Jul 2023
5	Exploring possible Problem Statements and Deriving Solutions	28 Jul 2023	31 Aug 2023
6	Project Plan Submission	4 Sep 2023	4 Sep 2023
System Design			
7	Design System Architecture Diagram	4 Sep 2023	11 Sep 2023
8	Document Use Cases & Diagrams	11 Sep 2023	18 Sep 2023

9	Define Technology Stack & Development tools selected	11 Sep 2023	18 Sep 2023
10	Front-end UI/UX design	18 Sep 2023	25 Sep 2023
Implementation			
11	Developing required Smart Contracts (User/Messaging/Finance/Storage)	18 Sep 2023	9 Oct 2023
12	Developing Storage System (IPFS)	2 Oct 2023	23 Oct 2023
13	Deploying Smart Contracts to Testnet (Sepolia)	23 Oct 2023	30 Oct 2023
14	Front-end Implementation (React)	30 Oct 2023	4 Dec 2023
15	Integration of Front-end with Back-end	4 Dec 2023	18 Dec 2023
16	Documentation of Implementations	18 Dec 2023	25 Dec 2023
Testing			
17	Unit Testing	25 Dec 2023	8 Jan 2024
18	Integration Testing	8 Jan 2024	15 Jan 2024
19	Documentation of Testing Phase	15 Jan 2024	22 Jan 2024
20	Final Clean-up of Interim Report Documentation	22 Jan 2024	29 Jan 2024
21	Interim Report Submission	29 Jan 2024	29 Jan 2024
Project Deliverables & Evaluation			
22	Code Cleanup	29 Jan 2024	19 Feb 2024
23	Documentation Cleanup	19 Feb 2024	25 Mar 2024
24	Final Report Submission	25 Mar 2024	25 Mar 2024
25	Editing Final Report	25 Mar 2024	19 Apr 2024
26	Amended Final Report Submission	19 Apr 2024	19 Apr 2024
27	Oral Presentation Preparation	19 Apr 2024	10 May 2024
28	Oral Presentation	10 May 2024	15 May 2024

Table 1: Work Breakdown Structure

1.6 REPORT ORGANISATION

The Report is organised into the following 7 chapters.

1. Introduction

   This chapter introduces the development project and its goals. Providing the background and motivation of the project, stating the problem the project addresses and the boundaries of the project, specifying what is in and out of the scope. The structure and development schedule is also outlined.
2. Literature Review

   This chapter surveys and reviews existing research, theories and application related to the development project, highlighting gaps and limitations in existing solutions.
3. Requirement Analysis

   This chapter identifies the functional and non-functional requirements of the project, documenting use cases and relevant diagrams.
4. Application Design

   This chapter defines the architecture and design of the application based on specific requirements, including the system architecture and User Interface (UI) design.
5. Implementation

   This chapters documents the development process of translating the design specifications into the actual application. It includes details about smart contract, backend and frontend implementations.
6. Testing

   This chapter depicts the process taken to validate and verify that the developed application meets the specific requirements using black box and white box testing.
7. Future Work & Conclusion

   This chapter discusses potential enhancements, future features and considerations for ongoing development of the application. Summary of the project's outcomes and the next steps for future development are also provided.

Chapter 2 : LITERATURE REVIEW

2.1 BLOCKCHAIN IN SECURE COMMUNICATION

2.1.1 GENERAL USAGE OF BLOCKCHAIN

There are many industries pursuing blockchain-based technology solutions to improve efficiency, streamline business processes and build trust between parties with little or no knowledge of each other. Based on Statista, forecasts suggests that spending on blockchain solutions will continue to grow in the coming years, reaching almost 19 billion U.S. dollars by 2024 [7]. This suggests that the need for privacy, data security solutions such as peer-to-peer decentralised systems have gone up and will continue to increase exponentially in the future.

In smart cities technologies, blockchain technology is also implemented for the secure communication between components in a smart city environment [8].

For instance, a secure framework for Electric Vehicles (EVs) synchronisation at Charging stations (CS) using smart contracts and IPFS based on a 5G wireless network is proposed. The primary goal is to ensure secure data storage and synchronize EV charging through the blockchain network. The combination of blockchain and IPFS enhances the security and cost-effectiveness of EV charging at the corresponding CS [9].

2.1.2 Soulbound Tokens For Digital Identity

Web3 today lack primitives to represent social identity, hence it has become fundamentally dependant on the very centralised web2 structures to support activities ubiquitous in the real economy, such as undercollateralized lending. To address this issue, Vitalik Buterin (Ethereum's Co-founder) along with Puja Ohlhaver and E. Glen Weyl proposed the concept of "Soulbound Tokens" (SBTs) [10].

SBTs are publicly verifiable and non-transferrable tokens that represents commitments, credentials and affiliations in the "Decentralised Society" (DeSoc) where "Souls" and communities come together bottom-up. "Souls" refer to accounts to which SBTs are permanently bound. For instance, a person may have a soul that stores SBTs that represents their educational credentials, employment history, where these SBTs can be "self-certified" just like in a CV. Implementations of SBTs include NFT ownership, job vacancies, gaming, verification of borrowing and lending and many more. Since authenticity cannot be purchased through SBTs, it fosters confidence among token holders and the general public to establish provenance and reputation.

2.1.3 ERC-735 Standard For Identity Management

The ERC-735 standard, created by Fabian Vogelsteller, is a standard on the Ethereum blockchain mainly used for issuing and managing identities through smart contracts [11]. The standard includes functions for adding, removing and holding of claims. Claims can be of several topics, from attestations of skills, qualifications to user personal data. Claims can be attested from third-parties or self-attested, providing a step forward in addressing KYC (Know Your Customer) issues in blockchain [12]. This provides a standardised way to manage and verify identities on the Ethereum blockchain. ERC-735 also contribute to enhancing the value of user's identities in the DeFi market. Users can add more elements to their identity such as personal information, introductions, and avatars, similar to popular social networking cites such as LinkedIn and X (Twitter). This also helps DApps better verify the identity of genuine users and create transparency in the use of their applications.

2.2 Existing Applications

2.2.1 Centralised Messaging Applications

Security measures that focuses on end-to-end encryption and user privacy, in popular messaging applications such as WhatsApp is evaluated.

The table below is a table of comparison between general centralised messaging applications and the proposed blockchain messaging application.

Aspect	General Messaging Application (e.g. WhatsApp)	Blockchain Messaging Application (ChainTalk)
Data Storage	Centralised storage on WhatsApp servers	Decentralised storage across Blockchain and IPFS
Ownership and Control	Controlled/Managed by WhatsApp/Meta	User-controlled data ownership and sovereignty
Security	Relies on central sever for security measures	Enhanced security though Blockchain's cryptographic technology
Data Integrity	Centralised data susceptible to manipulation	Immutable records ensures data integrity on the Blockchain
Interoperability	Closed ecosystem limited to WhatsApp users	Potential for cross-platform and Blockchain integration.

Table 2: Table of Comparison

2.2.2 BLOCKCHAIN MESSAGING APPLICATIONS: KEYBASE

Keybase is a centralized messaging application implemented on the blockchain for robust end-to-end encryption for secure communication [13]. The platform employs advanced encryption techniques, ensuring that messages remain confidential during transmission. However, a notable characteristic of Keybase is its reliance on centralized servers for message storage. While end-to-end encryption safeguards the content of messages, the fact that these messages are primarily stored on Keybase's centralized servers introduces a central point of vulnerability. This centralized storage approach may raise concerns related to data privacy and security.

Moreover, as a centralized messaging solution, Keybase faces limitations inherent in such systems, including potential risks of server outages and the need for users to trust the centralized entity managing their messages.

Chapter 3 : REQUIREMENT ANALYSIS

3.1 TARGET USERS

The target audience for ChainTalk the application encompasses a diverse group of individuals and entities seeking to enhance privacy, security and user control over their digital communications. This includes business professionals engaged in sensitive communications, individuals residing in regions with heightened data privacy concerns, and security conscious users seeking innovative solutions and many more. Moreover, ChainTalk also caters to those dissatisfied with existing messaging applications, offering an alternative that safeguards their digital data. Through a user-centric approach, ChainTalk aims to address the diverse needs of these audiences by providing features such as blockchain immutability, user data ownership and seamless user interface tailored to their preferences.

3.2 FUNCTIONAL REQUIREMENTS

3.2.1 PRODUCT FUNCTIONAL REQUIREMENT

1. The system shall allow users to be able to interact with the backend smart contract using the frontend web interface.
    1.1. Users with their Metamask wallet connected should be able to confirm or reject smart contract transactions via the Metamask browser extension.
    1.2. Users with their Metamask wallet connected should be able to view the total transaction fee being spent in the transaction.

2. The system shall provide secure user registration and authentication functionality.
    2.1. The user must connect their Metamask wallet to the Dapp.
    2.2. The user shall only submit an account registration request after their Metamask wallet is connected to the web application.
    2.3. The system must provide error prompts when the Email and password does not satisfy the criteria.
        2.3.1. Ethereum wallet address, Username and Email must satisfy the following conditions:
            2.3.1.1. Unique in the database
        2.3.2. Password must satisfy the following conditions:
            2.3.2.1. Have a length of at least 8 characters.

2.3.2.2. Contains both numeric and alphabetic characters.

2.4. The system must allow the user to register for an account securely with their Ethereum wallet address, Username, Email and Password.

2.4.1. Ethereum wallet address, Username and Email must satisfy the following conditions:

2.4.1.1. Unique in the database

2.4.2. The system will retrieve the unique Ethereum wallet address from the user's Metamask wallet connected.

2.4.3. Password string must satisfy the following conditions:

2.4.3.1. Have a length of at least 8 characters.

2.4.3.2. Contains both numeric and alphabetic characters.

2.5. The system must hash the Password string the user provides for storage on the blockchain.

2.6. The system shall tie the user's Username, Email and Password hash to their public Ethereum wallet address for storage on the blockchain.

2.7. The system shall create a unique non-transferrable Soulbound token when a user has been created successfully.

2.8. The Soulbound token must satisfy the following conditions:

2.8.1. Have a unique token ID.

2.8.2. Have a unique token URI.

2.9. The system shall create a new identity contract that is of ERC-735 standard when a user has been created successfully.

2.10. The system shall add the user's name, email, password hash, registration status, friend list and Soulbound token ID as claims into the new identity contract.

2.11. The system shall emit an event in the backend smart contract when a user has been created successfully.

2.12. The system shall authenticate the validity of the user by verifying the existence of a Soulbound token.

2.13. The system must allow the user to disconnect their wallets at any point in time.

3. The system must allow registered users to add friends with other registered users.

3.1. The system must provide error prompts if either of the users are not registered.

3.1.1. The system must retrieve the user information from the blockchain.

3.1.2. The system must ensure that both users exist in the blockchain.

3.2. The user must provide the Ethereum wallet address of the other registered user they wish to add as friend.

3.2.1. The system must ensure that the wallet address provided is valid.

3.2.1.1. The system shall search the blockchain for the existence of the wallet address.

3.3. The system is required to append the wallet address of newly added user to the initial user's friend list.

3.3.1. The system is required to append the wallet address of the initial user to the newly added user's friend list.

4. The system must allow registered users to send and receive messages among their friends.

4.1. The system must ensure that both users are already friends.

4.1.1. The system must retrieve the friend list of both users from the blockchain.

4.2. The system must retrieve all the user's previously added friends.

4.2.1. The system must retrieve the user's friend list from the blockchain based on user's Ethereum wallet address.

4.2.2. The system must display the user's friends on the web interface.

4.3. The user shall be able to send a message to the friend they selected.

4.3.1. The user must be able to select the friend they wish to send a message to on the web interface.

4.3.2. The user must be able to enter the content of the message in the input field.

4.3.3. The user must be able to click the send icon to send the message.

4.3.4. The system must be able to interact with the backend smart contract to send the message.

5. The system must allow registered users to login into their account and retrieve their information.

5.1. The user must connect their Metamask wallet to the Dapp.

5.1.1. The Ethereum wallet address retrieved from the connected wallet must be the one that the user used previously to register for an account.

5.1.2. The system must provide error prompt if the connected Metamask wallet does not contain an Ethereum wallet that was previously registered.

5.1.2.1. The system shall search the blockchain for the existence of the wallet address.

5.2. The system must display the Username of the user on the web interface.

5.2.1. The system must retrieve the user’s Username from the blockchain based on user’s Ethereum wallet address.

5.3. The system must retrieve all the user’s previously added friends.

5.3.1. The system must retrieve the user’s friend list from the blockchain based on user’s Ethereum wallet address.

5.3.2. The system must display the user’s friends on the web interface in a list view.

5.4. The system must retrieve all the user’s past messages.

5.4.1. The system must retrieve the IPFS CIDs from the blockchain based on user’s Ethereum wallet address.

5.4.2. The system must retrieve the full message content from the IPFS nodes using the IPFS CIDs.

5.4.3. The system must display the full messages retrieved on the web interface.

6. The system must allow registered users to update their friend list claims in their identity contract.

6.1. The system must the current user’s friend list.

6.2. The system must retrieve the current user’s identity contract address.

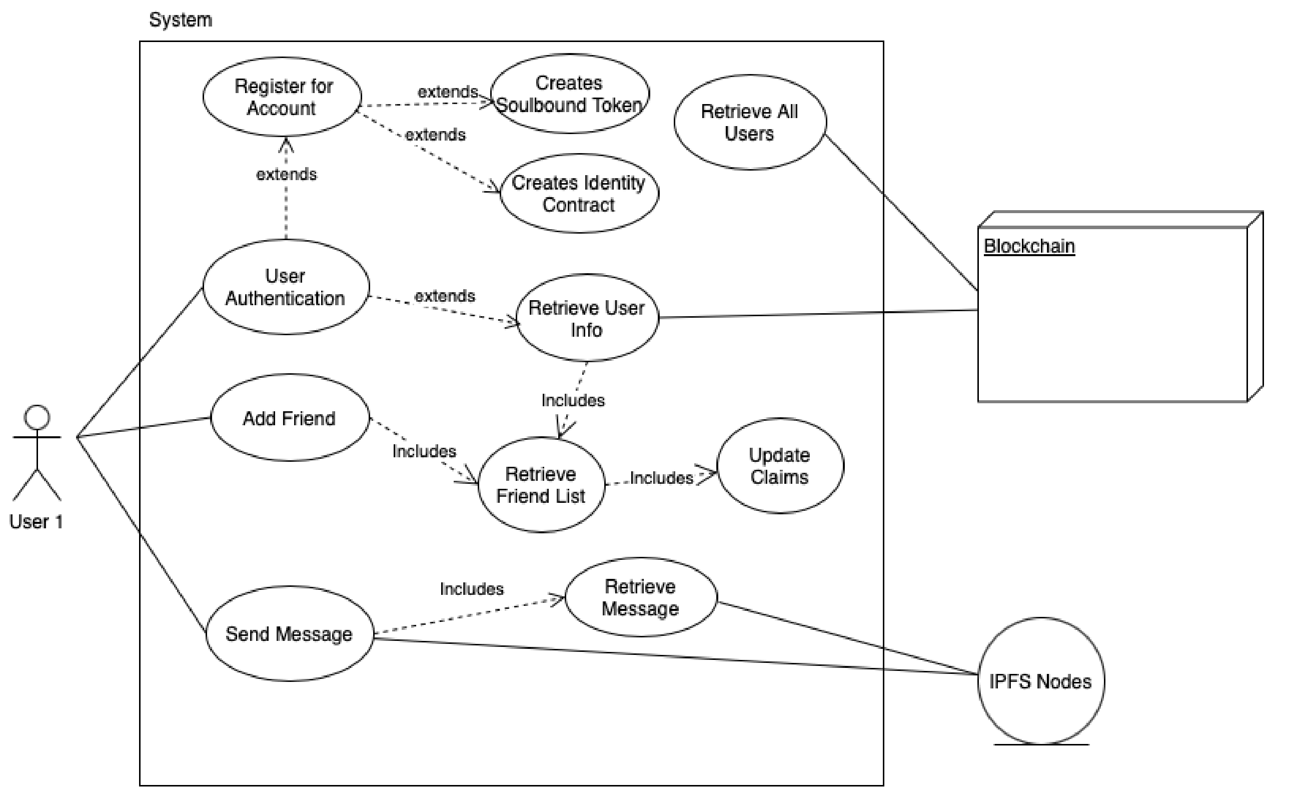
6.3. The system shall add the user’s new friend list as claims into the retrieved identity contract.

7. The system must provide dynamic webpage sizing.

7.1. Users with across various device sizes should be able to optimally view the webpage.

3.2.2 USE CASE DIAGRAM

The following diagram is a visual representation that illustrates the interactions between the actors, the system and the various use cases the system offers. The actors includes the users, the blockchain which forms the on-chain storage system and the IPFS nodes which forms the off-chain storage system.



Created using Drawio

Figure 2: Use Case Diagram

3.2.3 USE CASE DESCRIPTIONS

The following tables consist of the use cases implemented and its following descriptions.

Use Case ID:	UC001
Use Case Name:	User Authentication
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	Authenticate existing users.
Preconditions:	1. User has an account. 2. User has connected their Metamask wallet.
Postconditions:	The system will verify the validity of the user.
Priority:	High
Frequency of Use:	Every time a registered user enters the webpage.
Flow of Events:	1. The system directs a previously registered user to the homepage. 2. The system checks for the existence of a Soulbound token. 3. Upon successful validation, the system unlocks the user account. 4. The system directs the user to the homepage.
Alternative Flows:	1. If the user does not have an existing account. 2. Redirect to UC002: Register For Account
Exceptions:	The web interface will display an error message to the user if the Soulbound token does not exist. “Soulbound token does not exist. Please create an account.”
Includes:	-

Table 3: Use Case Description for UC001

Use Case ID:	UC002
Use Case Name:	Register For Account
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	User registers for a new account
Preconditions:	1. User has not registered for an account before. 2. User has connected their Metamask wallet. 3. User must have a valid Email address and Password that satisfies the conditions.
Postconditions:	1. New account is created and data is stored on blockchain. 2. The website redirects the user to the homepage.
Priority:	Medium
Frequency of Use:	Once per user in lifetime
Flow of Events:	1. User clicks on "Create Account" button. 2. The system redirects the user to the account creation page. 3. The system checks the blockchain if there is an existing Ethereum address being used for another user. 4. User types in a valid username, email and password. 5. User clicks on the "Submit" button. 6. The system prompts the user to "Confirm" or "Reject" the transaction. 7. The system validates the entries and directs the user to UC003 and UC004. 8. The system validates the entries and adds a new user record to the blockchain. 9. Upon successful transaction, User is redirected to the homepage.
Alternative Flows:	1. If invalid fields (e.g. Username, Email, Password) are entered, the web interface will display an error message. 2. The system will clear the fields and User page will reload.
Exceptions:	The web interface will display an error message to the user if any of the input fields are not filled in: 1. "Invalid Username entered" 2. "Invalid Email entered" 3. "Invalid Password entered"

Includes:	UC003: Creates Soulbound Token and UC004: Creates Identity Contract

Table 4: Use Case Description for UC002

Use Case ID:	UC003
Use Case Name:	Creates Soulbound Token
Created By:	Poon Yan Xin Melise
Date Created:	20/02/2024

Actor:	User, Blockchain
Description:	User registers for a new account and Soulbound token is created.
Preconditions:	1. User enters UC002: Register for account. 2. User has connected their Metamask wallet.
Postconditions:	1. Unique Soulbound token is created. 2. The website redirects the user to the homepage.
Priority:	Medium
Frequency of Use:	Once per user in lifetime
Flow of Events:	1. User enters UC002: Register for account. 2. The system creates a new Soulbound token with unique token ID and token URI. 3. The system adds the Soulbound token ID and Soulbound token as a new entry into the user record on the blockchain. 4. Upon successful transaction, User is redirected to the homepage.
Alternative Flows:	-
Exceptions:	-
Includes:	UC002: Register for account.

Table 5: Use Case Description for UC003

Use Case ID:	UC004
Use Case Name:	Creates Identity Contract
Created By:	Poon Yan Xin Melise
Date Created:	20/02/2024

Actor:	User, Blockchain
Description:	User registers for a new account and Identity contract is created.
Preconditions:	1. User enters UC002: Register for account. 2. User has connected their Metamask wallet.
Postconditions:	1. Identity contract is created. 2. The website redirects the user to the homepage.
Priority:	Medium
Frequency of Use:	Once per user in lifetime
Flow of Events:	1. User enters UC002: Register for account. 2. The system validates the entries entered by user and creates a new Identity contract with ERC-735 standard. 3. The system adds the entries as claims into the new Identity contract. 4. The system adds the Identity contract address as a new entry into the user record on the blockchain. 5. Upon successful transaction, User is redirected to the homepage.
Alternative Flows:	-
Exceptions:	-
Includes:	UC002: Register for account.

Table 6: Use Case Description for UC004

Use Case ID:	UC005
Use Case Name:	Retrieve User Info
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	User data is retrieved from the backend databases
Preconditions:	1. User has an account. 2. User has connected their Metamask wallet.
Postconditions:	The system will display retrieved user data on the homepage. This includes: 1. Username

	2. Friend List
Priority:	High
Frequency of Use:	Every time a registered user enters the webpage.
Flow of Events:	1. The system directs a previously registered user to the homepage. 2. The system retrieves the Username and Friend List from the blockchain based on User's Ethereum Address. 3. The system displays the retrieved data on the homepage.
Alternative Flows:	1. The system fails to retrieve user data, because user does not have an existing account. 2. Webpage will display an error message.
Exceptions:	The web interface will display the following error message to the user is a new user and has not created an account. "Please register for an account first."
Includes:	UC002: Register For Account

Table 7: Use Case Description for UC005

Use Case ID:	UC006
Use Case Name:	Add Friend
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	User adds another user as Friend.
Preconditions:	1. Both Users have an account. 2. User has connected their Metamask wallet. 3. Both Users are authenticated. 4. User is in the Chat page.
Postconditions:	1. Each User's Ethereum address is added to the other's friend list. 2. Appended friend list array is stored on the blockchain.
Priority:	Medium
Frequency of Use:	Every time a user wishes to add a new friend.
Flow of Events:	1. User clicks on "Add Friend" button.

	2. The system directs the user to the Add Friend Page. 3. User keys in the Ethereum address of another user. 4. User clicks on "Submit" button. 5. The system prompts the user to "Confirm" or "Reject" the transaction. 6. The system validates the entry and add the new address to the User's friend list array on the blockchain. 7. The system also adds the current user's address to the other user's friend list array on the blockchain. 8. Upon successful transaction, the system redirects the user back to the Chat Page.
Alternative Flows:	1. If invalid address is entered, the web interface will display an error message. a. The system will deem the address as invalid if the address belongs to a user that is not registered. 2. The system will clear the fields and User page will reload.
Exceptions:	The web interface will display an error message to the user if any the input fields is not filled in: "Invalid address entered"
Includes:	-

Table 8: Use Case Description for UC006

Use Case ID:	UC007
Use Case Name:	Retrieve Friend List
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	User friend list is retrieved from the backend databases
Preconditions:	1. User has an account. 2. User has connected their Metamask wallet. 3. User is authenticated. 4. User is in the Chat page.

Postconditions:	The system will display retrieved user friend list on the Chat Page.
Priority:	High
Frequency of Use:	Every time a registered user enters the webpage.
Flow of Events:	1. User enters the Chat page. 2. The system retrieves the Friend List from the blockchain based on User’s Ethereum Address. 3. The system displays the retrieved data on the Chat page.
Alternative Flows:	1. The system fails to retrieve user data, because user does not have any friends. 2. Webpage will display an error message.
Exceptions:	The web interface will display an error message to the user if the user does not have any friends. “Currently no friends.”
Includes:	-

Table 9: Use Case Description for UC007

Use Case ID:	UC008
Use Case Name:	Update Claims
Created By:	Poon Yan Xin Melise
Date Created:	20/02/2024

Actor:	User, Blockchain
Description:	User’s new friend list is added as claim to the Identity Contract.
Preconditions:	1. User has an account. 2. User has connected their Metamask wallet. 3. User is authenticated. 4. User has an Identity contract 5. User is in the Profile page.
Postconditions:	The system will update the friend list claim in the Identity Contract.
Priority:	Low
Frequency of Use:	Every time a registered user updates their friend list.
Flow of Events:	1. Users enters the Profile page.

	2. The User clicks on “Update” button. 3. The system retrieves the Friend List from the blockchain based on User’s Ethereum Address. 4. The system retrieves the user specific Identity contract address from the blockchain based on User’s Ethereum Address. 5. The system adds the Friend List retrieved as new claims to the Identity contract.
Alternative Flows:	1. The system fails to retrieve user data, because user does not have any friends. 2. Webpage will display an error message.
Exceptions:	The web interface will display an error message to the user if the user does not have any friends. “Currently no friends.”
Includes:	-

Table 10: Use Case Description for UC008

Use Case ID:	UC009
Use Case Name:	Retrieve All Users
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain
Description:	All user data is retrieved from the Blockchain to be displayed on the All Users page.
Preconditions:	1. User has an account. 2. User has connected their Metamask wallet. 3. User is authenticated. 4. User is in the All Users page.
Postconditions:	The system will display all registered Users on the All Users Page.
Priority:	High
Frequency of Use:	Every time a registered user enters the webpage.
Flow of Events:	1. User enters the All Users page.

	2. The system retrieves the All Users struct from the blockchain. 3. The system displays all the retrieved data on the All Users page. This data includes: 1. Username 2. User Ethereum address. 4. Users can choose to click on "Add Friend" to add the User as friend. 5. User enters flow of UC004.
Alternative Flows:	1. The system fails to retrieve user data, if there are no registered users. 2. Webpage will display an error message.
Exceptions:	The web interface will display an error message to the user if there are no registered users. "Currently no registered Users"
Includes:	UC004: Add Friend

Table 11: Use Case Description for UC009

Use Case ID:	UC010
Use Case Name:	Retrieve Message
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain, IPFS Nodes
Description:	System retrieves all past messages to display to the User.
Preconditions:	1. User has connected their Metamask wallet. 2. User is authenticated. 3. User is in the Chat page. 4. Both Users are already friends.
Postconditions:	The webpage displays all the previously exchanges messages between 2 users.
Priority:	High
Frequency of Use:	Every time the user clicks on a Chat in the Chat page
Flow of Events:	1. User clicks on the User they wishes to chat with on the left panel.

	2. The system retrieves the messages, including the IPFS hash of the message from the blockchain based on the User's Ethereum Address. 3. The system retrieves the full message content from the IPFS nodes using the IPFS CIDs. 4. The system displays all the past messages exchanged in the Chat field.
Alternative Flows:	1. The system fails to retrieve message if there are no previous messages exchanged.
Exceptions:	The web interface will display an error message to the user if no messages exist. "No previous messages"
Includes:	-

Table 12: Use Case Description for UC010

Use Case ID:	UC011
Use Case Name:	Send Message
Created By:	Poon Yan Xin Melise
Date Created:	10/01/2024

Actor:	User, Blockchain, IPFS Nodes
Description:	User sends a message to another User.
Preconditions:	1. User has connected their Metamask wallet. 2. User is authenticated. 3. User is in the Chat page. 4. Both users are already friends.
Postconditions:	The User successfully send a message to another User.
Priority:	High
Frequency of Use:	Every time a registered user wishes to send a message.
Flow of Events:	1. User clicks on the User they wishes to chat with on the left panel.

	2. If there are existing messages exchanged between the users, the system retrieves these messages (UC007: Retrieve Message). 3. User keys in the content of the new message in the input field. 4. User clicks on “Send” icon on the right side of the input field. 5. The system prompts the user to “Confirm” or “Reject” the transaction. 6. The system validates the message content and adds the new message to the IPFS nodes. 7. The system receives the IPFS CIDs of the message from the IPFS nodes. 8. The system adds the new message containing the IPFS CIDs to the blockchain. 9. Upon successful transaction, User is redirected to the Chat page. 10. The webpage displays the new message in the existing messages field.
Alternative Flows:	-
Exceptions:	-
Includes:	UC010: Retrieve Message.

Table 13: Use Case Description for UC011

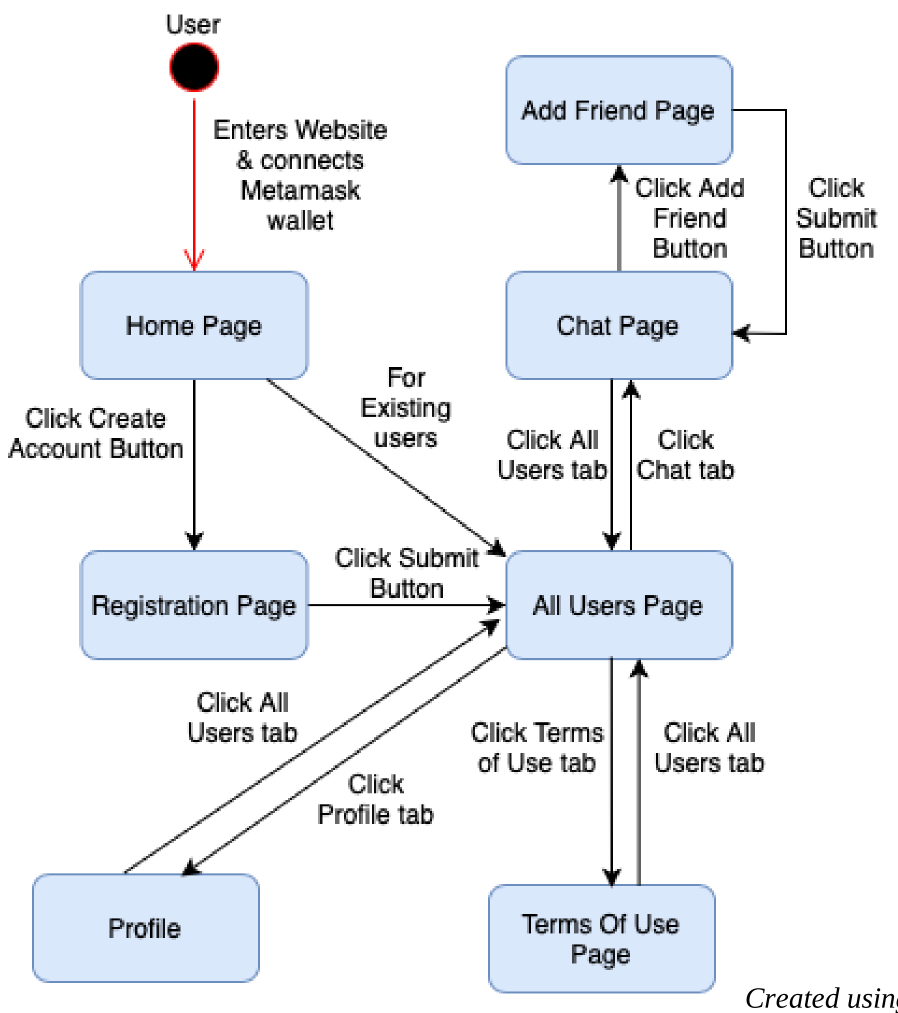
3.2.4 DIALOG MAP

The following dialog map depicts the architecture of the user interface design, outlining the flow of user interactions with the application. It depicts how the user will navigate through the system and interact with the various components.

The user first enters the website and connects their Metamask wallet to the site. Upon doing so, the system will direct the user to the homepage.

If the User is a new User, they can click on “Create Account” button to create a new account. Else, existing Users will be directed to the All Users page.

Users will be able to navigate to the different pages (e.g. All Users, Chat, Profile, Terms Of Use) to perform different functionalities with the aid of the in-built Navigation Bar.



Created using Drawio

Figure 3: Dialog Map

3.3 Non-functional Requirements

Category	Description
Performance	1. The time taken for the website to load shall not take more than 5 seconds. 2. The time taken for a smart contract interaction shall not take more than 15 seconds. 3. The time taken to re-render a new page shall not take more than 5 seconds. 4. For displaying of messages, upon user refresh, the system should be able to display the new messages within 10 seconds of receiving it.
Scalability	1. The website should be able to scale up to accommodate an increase in user traffic without any delay in performance or response time. 2. The website should be able to scale from sending 1 message per second to multiple messages per second by a user.
Portability	1. The website should be fully functional on all major browsers on localhost, including Chrome, Safari, Edge, Firefox with consistent layout and performance. 2. The website should maintain a dynamic user experience across various screen sizes and resolutions.
Compatibility	1. The frontend web application should be interoperable with the backend smart contracts and the IPFS nodes for storage and retrieval of data.

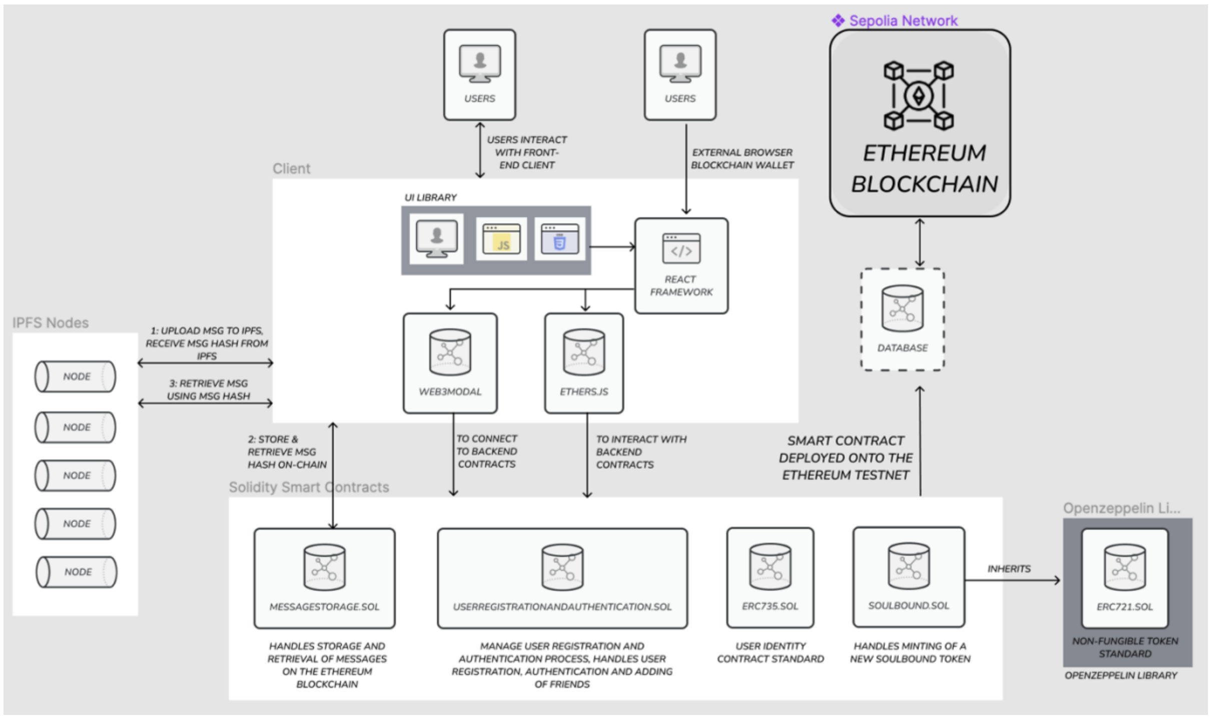
	2. The backend smart contracts should be interoperable with the blockchain for storage and retrieval of data.
Reliability	1. The system must perform without any failure during 95% of the use cases in a week. 2. The system should be able to handle and prompt users in instances of errors.
Usability	1. The error rate of users registering for an account must not exceed 10 percent. 2. Users should be able to search and chat with their friends within 3 clicks from the homepage. 3. The most frequently used feature (chat) must be easily accessible and reached with minimal navigation on the user's end.
Security	1. The website must conceal the password while typing it. 2. All user data and message CIDs should be stored securely on the blockchain.

Table 14: Non-functional Requirements

Chapter 4 : APPLICATION DESIGN

4.1 OVERALL APPLICATION ARCHITECTURE

The diagram represents the design and components used within the system.



Sepolia Network
ETHEREUM BLOCKCHAIN
USERS
USERS
USERS INTERACT WITH FRONT-END CLIENT
EXTERNAL BROWSER BLOCKCHAIN WALLET
Client
UI LIBRARY
JS
REACT FRAMEWORK
DATABASE
IPFS Nodes
1: UPLOAD MSG TO IPFS, RECEIVE MSG HASH FROM IPFS
3: RETRIEVE MSG USING MSG HASH
NODE
NODE
NODE
NODE
NODE
WEB3MODAL
ETHERS.JS
2: STORE & RETRIEVE MSG HASH ON-CHAIN
TO CONNECT TO BACKEND CONTRACTS
TO INTERACT WITH BACKEND CONTRACTS
SMART CONTRACT DEPLOYED ONTO THE ETHEREUM TESTNET
Solidity Smart Contracts
MESSAGESTORAGE.SOL
USERREGISTRATIONANDAUTHENTICATION.SOL
ERC735.SOL
SOULBOUND.SOL
INHERITS
Openzeppelin Li...
ERC721.SOL
NON-FUNGIBLE TOKEN STANDARD
OPENZEPPELIN LIBRARY
HANDLES STORAGE AND RETRIEVAL OF MESSAGES ON THE ETHEREUM BLOCKCHAIN
MANAGE USER REGISTRATION AND AUTHENTICATION PROCESS, HANDLES USER REGISTRATION, AUTHENTICATION AND ADDING OF FRIENDS
USER IDENTITY CONTRACT STANDARD
HANDLES MINTING OF A NEW SOULBOUND TOKEN

Created using Figma

Figure 4: Overall System Architecture Diagram

4.1.1 ETHEREUM BLOCKCHAIN - SEPOLIA

As this project is developed for educational purposes, the Ethereum Testnet Sepolia will be utilised in its development. Ethereum is a decentralised blockchain platform that establishes peer-to-peer network with smart contract functionality [14].

The Smart Contracts will be developed in Solidity, and deployed onto the Sepolia Testnet.

4.1.2 INTERPLANETARY FILE SYSTEM (IPFS)

IPFS is a set of composable, peer-to-peer protocols for addressing, routing, and transferring content-addressed data in a decentralised file system [15]. In this development project, the storage and retrieval of data from the file system is used. IPFS facilitates efficient content retrieval through its content-addressed model, enhancing the speed and reliability of message delivery. The decentralised nature of IPFS aligns seamlessly with the principles of our blockchain-based application, fostering a more robust and censorship-resistant communication environment. Additionally, IPFS contributes to reduced on-chain storage to promote efficiency of the sending of messages.

In this proposed solution, IPFS will be used for off-chain storage of user' message. Only the IPFS Content Identifiers (CIDs) will be stored in the MessageStorage contract on-chain. As blockchain size limits are small, it will incur high transactional costs if we store all the message data on-chain. Hence, using IPFS to store these data off-chain will reduce transactional cost for the sending of messages.

4.1.3 OPENZEPPELIN LIBRARY (OPEN-SOURCE)

OpenZeppelin is an open-sourced library for secure smart contract development [16]. In this development project, the ERC-721 non-fungible token standard implementation is inherited from the OpenZepplin library to form the Soulbound token.

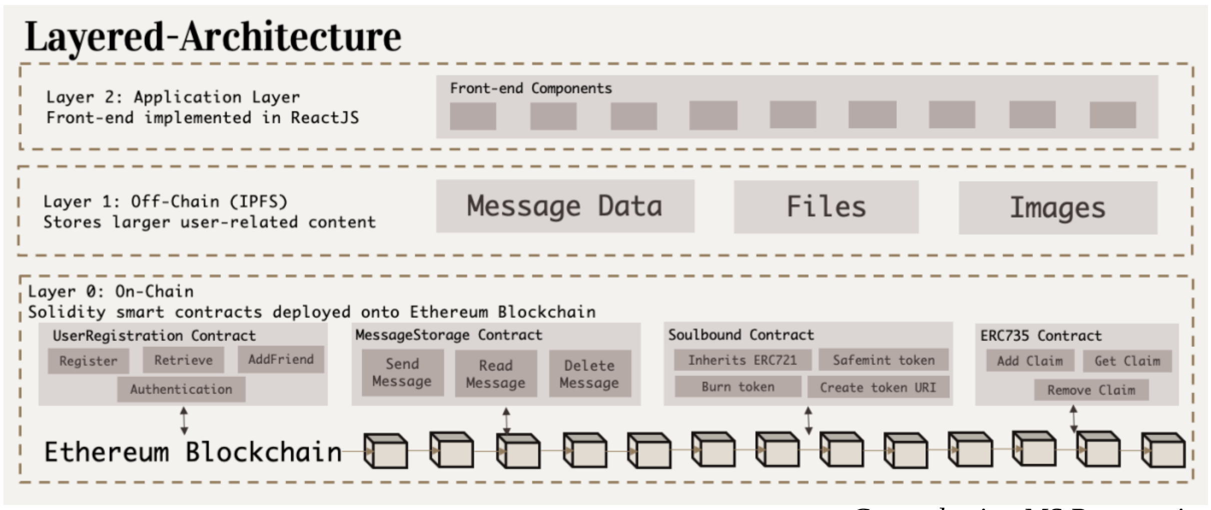
4.1.4 SOLIDITY SMART CONTRACTS

The 4 Solidity smart contracts implement the relevant logic and generate a chain of transaction records in the blockchain system. These contracts work together to form the core back-end functionalities of ChainTalk.

4.1.5 CLIENT

The front-end Client consist of the UI component library which forms the react framework for development. It also consist of Web3Model to connect to user's Ethereum wallets and Ethers.js to interact with the Ethereum blockchain.

4.2 LAYERED ARCHITECTURE



Created using MS Powerpoint

Figure 5: Layered Architecture

The ChainTalk application is built upon a total of 3 different layers forming the layered architecture that interacts cohesively with each other.

Layer 0: On-Chain

The base layer serves as the underlying infrastructure, providing the backbone for decentralised consensus, immutability, security and smart contract functionality.

It contains the 4 solidity smart contracts that are deployed onto the Ethereum blockchain. The solidity smart contracts handles user registration, authentication and storing of metadata and IPFS CIDs on the Ethereum blockchain. Ethereum addresses directly identify users within these smart contracts.

Layer 1: Off-chain (IPFS)

Positioned as an intermediate layer of this architecture, the storage protocol utilises the IPFS nodes for off-chain storage of larger user's message data that does not need to reside on the Ethereum blockchain to promote efficiency.

Layer 2: Application Layer

Built on top of the storage protocol, the messaging application, ChainTalk, interacts with the smart contracts and off-chain storage via IPFS for message storage, retrieval and management.

4.2.1 BENEFITS OF LAYERED ARCHITECTURE

1. Abstraction and Modularity: Easier development and maintenance of each layer independently.
2. Decentralisation and Efficiency:
    a. Blockchain for on-chain storage: Ensures decentralisation storage of data.
    b. IPFS for off-chain storage: Optimises blockchain efficiency, reduces on-chain transactional load.
3. Reusability and Scalability: Enables future enhancements to each layer without significantly affecting other layers.
4. Security: Secure storage of encrypted message hashes for tamper-resistant records.

4.3 Frontend Architecture

The frontend of the application is built using ReactJS. This architecture is designed using various reusable components to maximise scalability. Ethers.js is also used for the development of the Dapp to enable interaction of the frontend with the Ethereum blockchain.

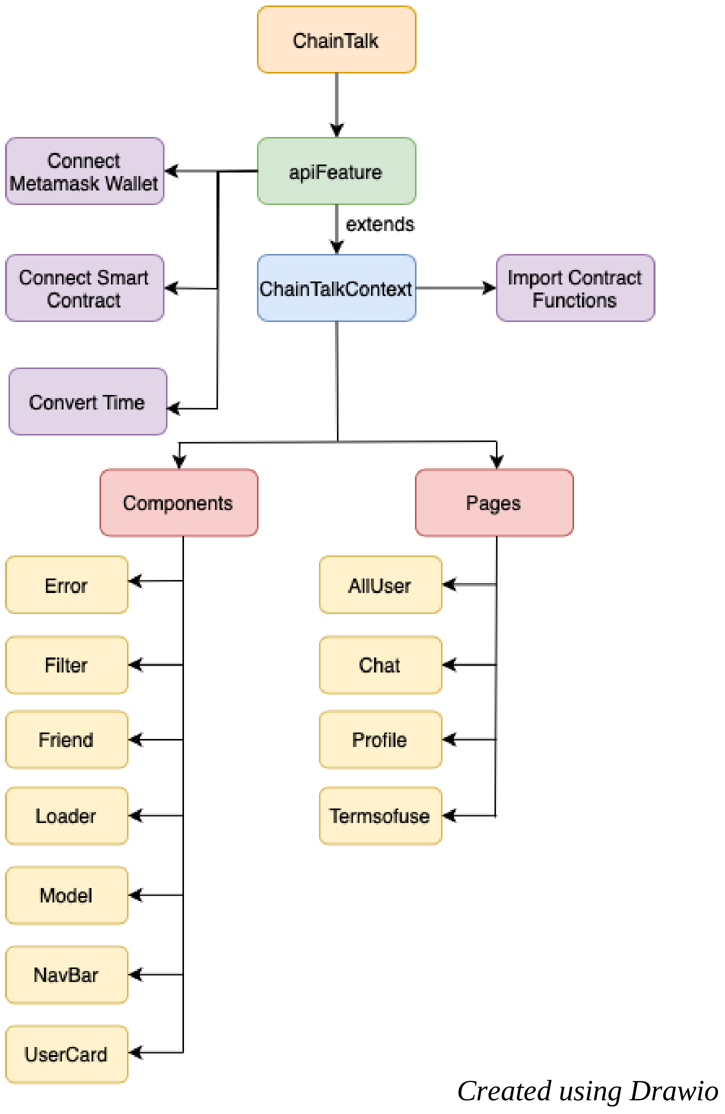


Figure 6: Frontend Architecture

apiFeature.js uses the Ethers.js JavaScript library to interact with the backend deployed smart contracts in order to send transactions and retrieve data from the blockchain. It also includes functions for users to connect their Metamask wallets and converts current time for future use by other components.

ChainTalkContext.js imports the connecting of Metamask wallet and backend contract interaction functions from apiFeature.js. Using these imported functions, we are able to call the functions implemented in the backend smart contracts for usage in the frontend interface.

Components consist of 7 different components that imports the backend smart contract functions from ChainTalkContext.js. These components are imported by pages.

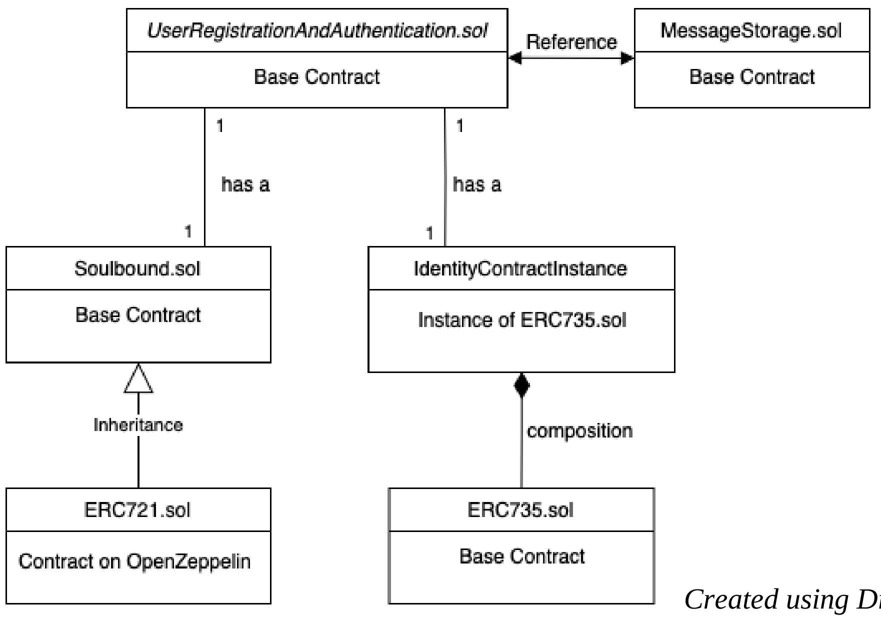
Pages consist of 4 different pages that imports the backend smart contract functions from ChainTalkContext.js. These pages interlink with each other to form the complete frontend user interface.

4.4 BACK-END/SMART CONTRACT ARCHITECTURE

The backend of ChainTalk is developed using smart contracts in Solidity that are deployed onto the Ethereum blockchain. Solidity is an object-oriented programming language used for implementing smart contracts on blockchain.

In this project, a total of four smart contracts are developed:

- UserRegistrationAndAuthentication.sol
- MessageStorage.sol
- Soulbound.sol
- ERC735.sol



Created using Drawio

Figure 7: Backend Smart Contract Architecture

The *UserRegistrationAndAuthentication* contract and *MessageStorage* contract has an association relationship where the contracts interact with each other to retrieve user information and send messages.

Each User registered from *UserRegistrationAndAuthentication.sol* has a Soulbound token (SBT) created from *soulbound.sol*. The *soulbound.sol* contract inherits functions from *ERC721.sol* contract from OpenZeppelin.

Each User registered from *UserRegistrationAndAuthentication.sol* also has an instance of Identity Contract. The identity contract instance has a composition relationship with *ERC735.sol*, whereby the identity contract is derived from *ERC735.sol*.

User contract stores the data of all users that exist in the system. When a new user is created, a unique non-transferrable SBT will be minted and an Identity contract derived from ERC735.sol will be created.

Message Storage contract manages storage and retrieval of all messages that exist in the system. This contract employs a modular approach, separating user detailed specific messages from a more summarised global view, and provides functions for sending, retrieving, and managing messages within a transparent and immutable environment.

Soulbound contract manages the minting of a SBT. To ensure the authenticity of a user, a soulbound token (SBT) is deployed for every registered user. SBT cannot be transferred to another user and will remain as a form of authentication measure for each user.

ERC735 contract provides the standard for the creation of an identity contract for each user. User’s attributes such as their name, email, password hash, registration status, friend list and SBT token ID are added into the identity contract as claims. This aims to enhance the dynamism of a user’s identity, ensuring that each user possesses a digital identity within the blockchain ecosystem.

4.5 APPLICATION NAME AND LOGO

The application, “ChainTalk” derives its name from the concept of facilitating communication via blockchain technology.

The design for the application logo is inspired from the interconnected nature fostered by the communication platform.

Below is an image showcasing the finalised logo design for the application.

Figure 8: Application Logo

Chapter 5 : IMPLEMENTATION

5.1 TECHNOLOGY STACK

The following are the software and their versions used for the development of ChainTalk.

Name (Version)	Description
Solidity (v0.8.21)	Solidity is an object-oriented programming language for the implementation of the smart contracts to be deployed on the Ethereum blockchain. It is designed to target the Ethereum Virtual Machine (EVM).
Hardhat (v2.9.3)	Hardhat is a local Ethereum development environment that is used to deploy, run tests and debug the smart contracts during development.
Ethers (v6.1.0)	Ethers.js is a JavaScript library for interaction with the Ethereum blockchain. It facilitates tasks such as creating and managing Ethereum wallets, sending transactions, querying blockchain data, and interacting with smart contracts in a user-friendly manner.
Chai (v4.2.0)	Chai is a JavaScript testing library commonly used in conjunction with testing frameworks like Mocha.
Mocha (v10.2.0)	Mocha is a JavaScript testing framework commonly used for both front-end and back-end testing. Mocha can be paired with assertion libraries like Chai to make test assertions.
ReactJS (v18.0.0)	ReactJS is a JavaScript library that follows a component-based architecture, allowing developers to create reusable UI components that update dynamically in response to data changes.

Web3Modal (v1.9.9)	Web3modal is a JavaScript library that simplifies the process of integrating and managing web3 providers (e.g. MetaMask)
OpenZeppelin (v4.0.0)	OpenZeppelin is an open-sourced library for building secure smart contract in the Solidity programming language.

Table 15: Software Used and Descriptions

The following are the software tools and its description for the development of ChainTalk

Name	Description
Visual Studio Code	Visual Studio Code (VS Code) is a free and open-source source-code editor developed by Microsoft. It supports a wide range of programming languages such as Solidity and JavaScript for the development of this project.
RemixIDE	Remix IDE is a web-based integrated development environment (IDE) for Ethereum smart contract development. Remix IDE provides a user-friendly interface for writing, testing, and deploying smart contracts on the Ethereum blockchain.
Metamask Chrome Extension	MetaMask is a popular browser extension for Chrome (and other browsers) that serves as a cryptocurrency wallet and gateway to decentralized applications (DApps) on the Ethereum blockchain.
Google Chrome Browser	Google Chrome is a widely-used web browser developed by Google.

Git	Git is a distributed version control system widely used for tracking changes in source code during software development.

Table 16: Software Tools used and Description

5.2 SMART CONTRACT IMPLEMENTATION

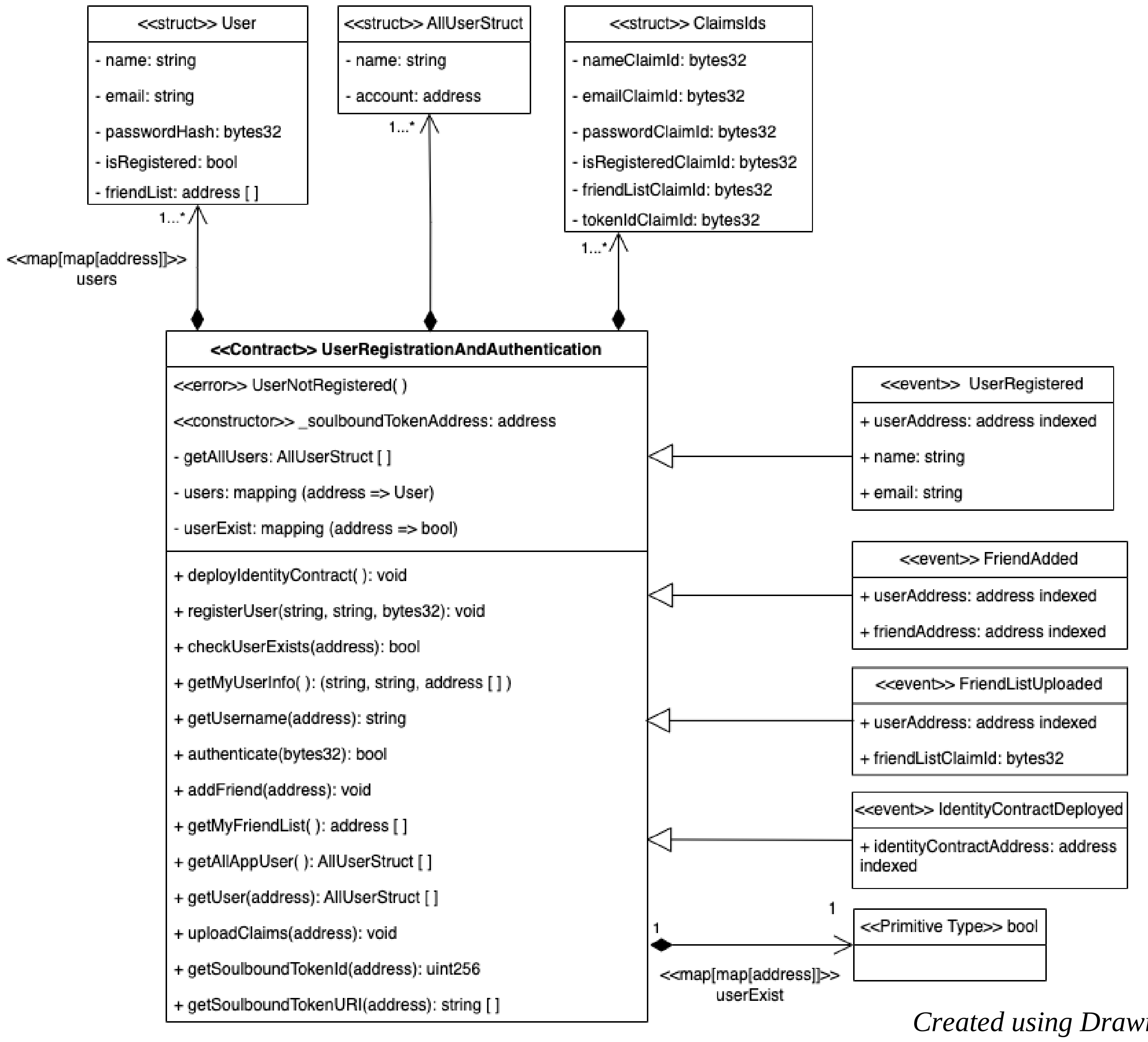
There is a total of 4 contracts implemented: *UserRegistrationAndAuthentication.sol*, *MessageStorage.sol*, *Soulbound.sol, ERC735.sol*. Hardhat is used for the deployment of the contracts to localhost and for command line interactions with the contract.

5.2.1 USER REGISTRATION CONTRACT

The user registration contract stores the data of all users that exist in the system. When a new user is created, their data will be stored in the User struct that will be mapped to their Ethereum addresses for easy retrieval by other functions. Their name and address will also be added in another struct that stores the data of all users in the system for display in the frontend interface. For this contract, multiple events are emitted to notify the frontend user interface to reflect the changes that occurred in the blockchain.

The full *UserRegistrationAndAuthentication.sol* code can be found in the Github Repository in Appendix A.

The below diagram provides a structural overview of the contract, including the data structures and key variables used.



<<struct>> User
- name: string
- email: string
- passwordHash: bytes32
- isRegistered: bool
- friendList: address [ ]
<<struct>> AllUserStruct
- name: string
- account: address
<<struct>> ClaimsIds
- nameClaimId: bytes32
- emailClaimId: bytes32
- passwordClaimId: bytes32
- isRegisteredClaimId: bytes32
- friendListClaimId: bytes32
- tokenIdClaimId: bytes32
1...*
1...*
1...*
<<map[map[address]]>>
users
<<Contract>> UserRegistrationAndAuthentication
<<error>> UserNotRegistered( )
<<constructor>> _soulboundTokenAddress: address
- getAllUsers: AllUserStruct [ ]
- users: mapping (address => User)
- userExist: mapping (address => bool)
+ deployIdentityContract( ): void
+ registerUser(string, string, bytes32): void
+ checkUserExists(address): bool
+ getMyUserInfo( ): (string, string, address [ ])
+ getUsername(address): string
+ authenticate(bytes32): bool
+ addFriend(address): void
+ getMyFriendList( ): address [ ]
+ getAllAppUser( ): AllUserStruct [ ]
+ getUser(address): AllUserStruct [ ]
+ uploadClaims(address): void
+ getSoulboundTokenId(address): uint256
+ getSoulboundTokenURI(address): string [ ]
<<event>> UserRegistered
+ userAddress: address indexed
+ name: string
+ email: string
<<event>> FriendAdded
+ userAddress: address indexed
+ friendAddress: address indexed
<<event>> FriendListUploaded
+ userAddress: address indexed
+ friendListClaimId: bytes32
<<event>> IdentityContractDeployed
+ identityContractAddress: address indexed
1
1
<<Primitive Type>> bool
<<map[map[address]]>>
userExist

Created using Drawio

Figure 9: UserRegistration Contract Class Diagram

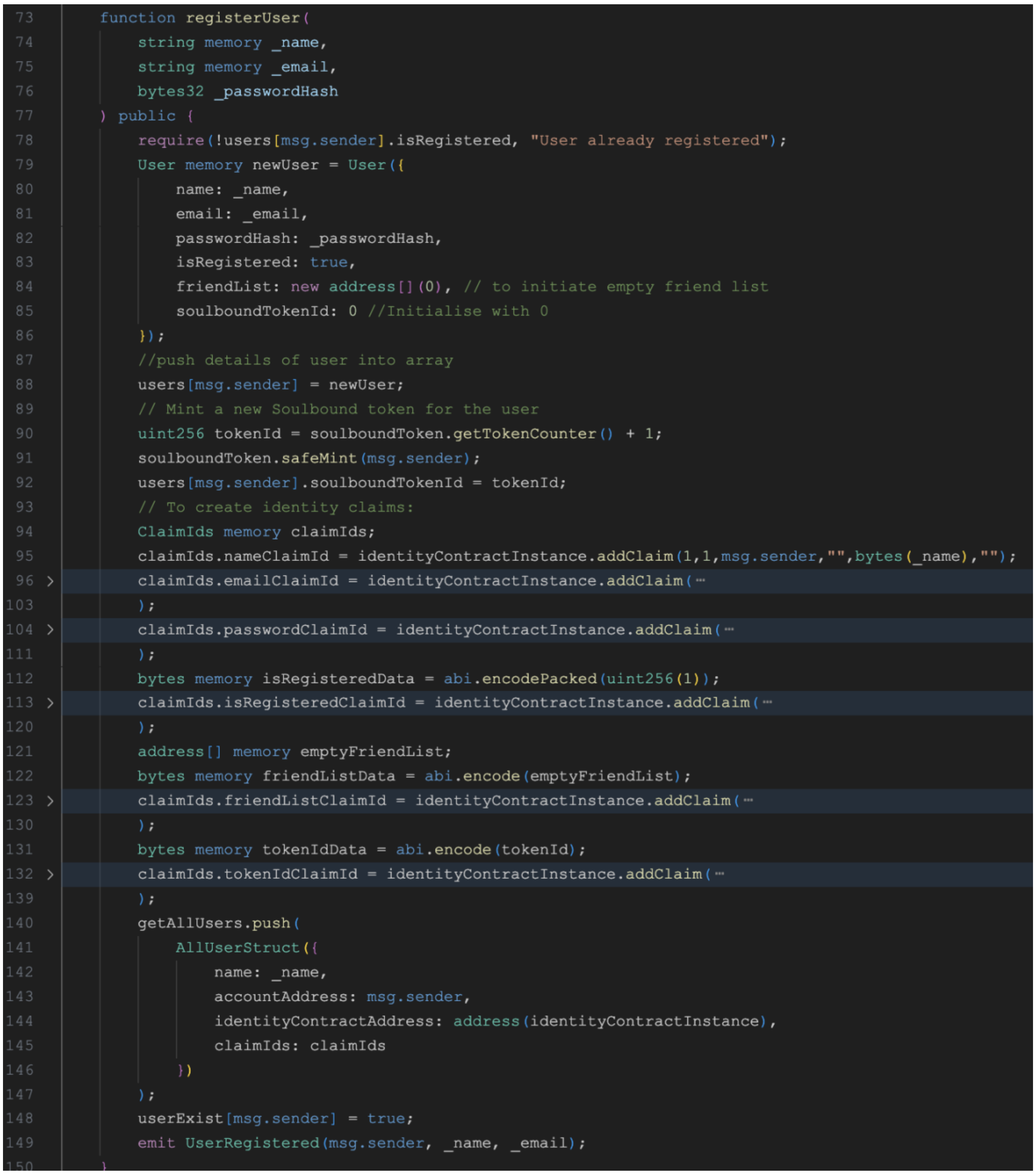
The following includes a more detailed explanation of the structs and variables implemented in the User Registration smart contract.

Variable Name	Variable Type	Description
User	Struct	Stores the data of each user, including the user's name, email, passwordHash, friendList and whether the user is registered is stored.
AllUserStruct	Struct	Stores all registered users of the application, including their name and their metamask address.
ClaimsId	Struct	Stores the claims id of each user, including the nameClaimId, emailClaimId, passwordClaimsId, isRegisteredClaimId, friendListClaimId and tokenIdClaimId.
users	mapping(address => User)	User's Ethereum address is a unique key that corresponds to the struct which contains the data of that particular user. Mapping enables quick and efficient retrieval of user data based on their addresses.
userExist	mapping(address => bool)	User's Ethereum address is a unique key that corresponds to a Boolean value. This mapping ensures quick and efficient verification of the existence of the user based on their addresses.

Table 17: User Registration Contract Variables

The following describes the registerUser function of the smart contract.

registerUser



```
    function registerUser(
        string memory _name,
        string memory _email,
        bytes32 _passwordHash
    ) public {
        require(!users[msg.sender].isRegistered, "User already registered");
        User memory newUser = User({
            name: _name,
            email: _email,
            passwordHash: _passwordHash,
            isRegistered: true,
            friendList: new address[](0), // to initiate empty friend list
            soulboundTokenId: 0 //Initialise with 0
        });
        //push details of user into array
        users[msg.sender] = newUser;
        // Mint a new Soulbound token for the user
        uint256 tokenId = soulboundToken.getTokenCounter() + 1;
        soulboundToken.safeMint(msg.sender);
        users[msg.sender].soulboundTokenId = tokenId;
        // To create identity claims:
        ClaimIds memory claimIds;
        claimIds.nameClaimId = identityContractInstance.addClaim(1,1,msg.sender,"",bytes(_name),"");
        claimIds.emailClaimId = identityContractInstance.addClaim(⋯
        );
        claimIds.passwordClaimId = identityContractInstance.addClaim(⋯
        );
        bytes memory isRegisteredData = abi.encodePacked(uint256(1));
        claimIds.isRegisteredClaimId = identityContractInstance.addClaim(⋯
        );
        address[] memory emptyFriendList;
        bytes memory friendListData = abi.encode(emptyFriendList);
        claimIds.friendListClaimId = identityContractInstance.addClaim(⋯
        );
        bytes memory tokenIdData = abi.encode(tokenId);
        claimIds.tokenIdClaimId = identityContractInstance.addClaim(⋯
        );
        getAllUsers.push(
            AllUserStruct({
                name: _name,
                accountAddress: msg.sender,
                identityContractAddress: address(identityContractInstance),
                claimIds: claimIds
            })
        );
        userExist[msg.sender] = true;
        emit UserRegistered(msg.sender, _name, _email);
```

Figure 10: Code for RegisterUser Function

1. The function takes in new user’s name, email and passwordHash as input parameter to register a new user.
2. Ensures that user is not registered before the execution of the function.
3. A new entry will be created to be assigned into the *users* mapping with msg.sender as key.

4. Mints a new soulbound token for the user and assigns its ID to the user’s struct.
5. Created several identity claims for the user. Claims include: Name, Email, Password, Registration status, Friend list, Token ID.
6. Adds user’s details including claimIDs to the *getAllUsers* array.
7. Boolean value in *userExist* array is also updated to true.
8. Emits an event when the function is successful.

The following table describes the rest of the functions the contract contains.

Function	Description
deployIdentityContract	Instantiates a new instance of the ERC753 contract and assigns it to the \`identityContractInstance\` variable. Emits an event, passing the address of the newly deployed identity contract instance as an argument.
getMyUserInfo	Returns all user info (name, email and friendList) from the *users* array based on user’s address.
checkUserExists	Checks whether a user already has an account by returning the Boolean value from the *userExist* array of the user.
getUsername	Ensures that user is registered before the execution of the function. Returns Username from *users* array based on user’s address.
authenticate	Ensures that user is registered before the execution of the function. Authenticates a User by checking whether a soulbound token id exist for the user by retrieving tokenID from the users struct. Returns a Boolean value if the check passes.
addFriend	Takes in a user address as input parameter. Ensures that user is registered before the execution of the function. Ensures that address of the user is not the same as the provided friend’s address. Checks that user address passed in corresponds to a registered user. Reverts a custom error, *UserNotRegistered( ),* if user address passed in does not correspond to an existing user.

	Declares a Boolean variable, *isFriend*, to be false and iterate over user’s friend list and checks if the current element in the friend list is equals to the provided friend’s address. If condition is true, set *isFriend* variable to true. If *isFriend* variable is set to true, an exception is raised as the two users are already friends. Push the user’s address into the *friendList* entry of the *users* array and vice versa for the friend. Emits an event, *FriendAdded*, when the function is successful.
getMyFriendList	Ensures that user is registered before the execution of the function. Returns the *friendList* of the user from the *users* array.
getAllAppUser	Returns the *getAllUsers* struct which contains the array of all users.
uploadClaims	Takes in a user address as input parameter. Ensures that user is registered before the execution of the function. Retrieves the user’s current friend list from the user struct Adds the new friend list as claims into the identity contract. Emits an event *FriendListUploaded*, when function is successful. Updates the claim ID in the user’s ClaimsIds struct with the new claim ID.
getSoulboundTokenId	Takes in a user address as input parameter. Ensures that user is registered before the execution of the function. Returns a uint256 consisting of the Soulbound Token ID.
getSoulboundTokenURI	Takes in a user address as input parameter. Ensures that user is registered before the execution of the function. Returns a string consisting of the Soulbound Token URI.

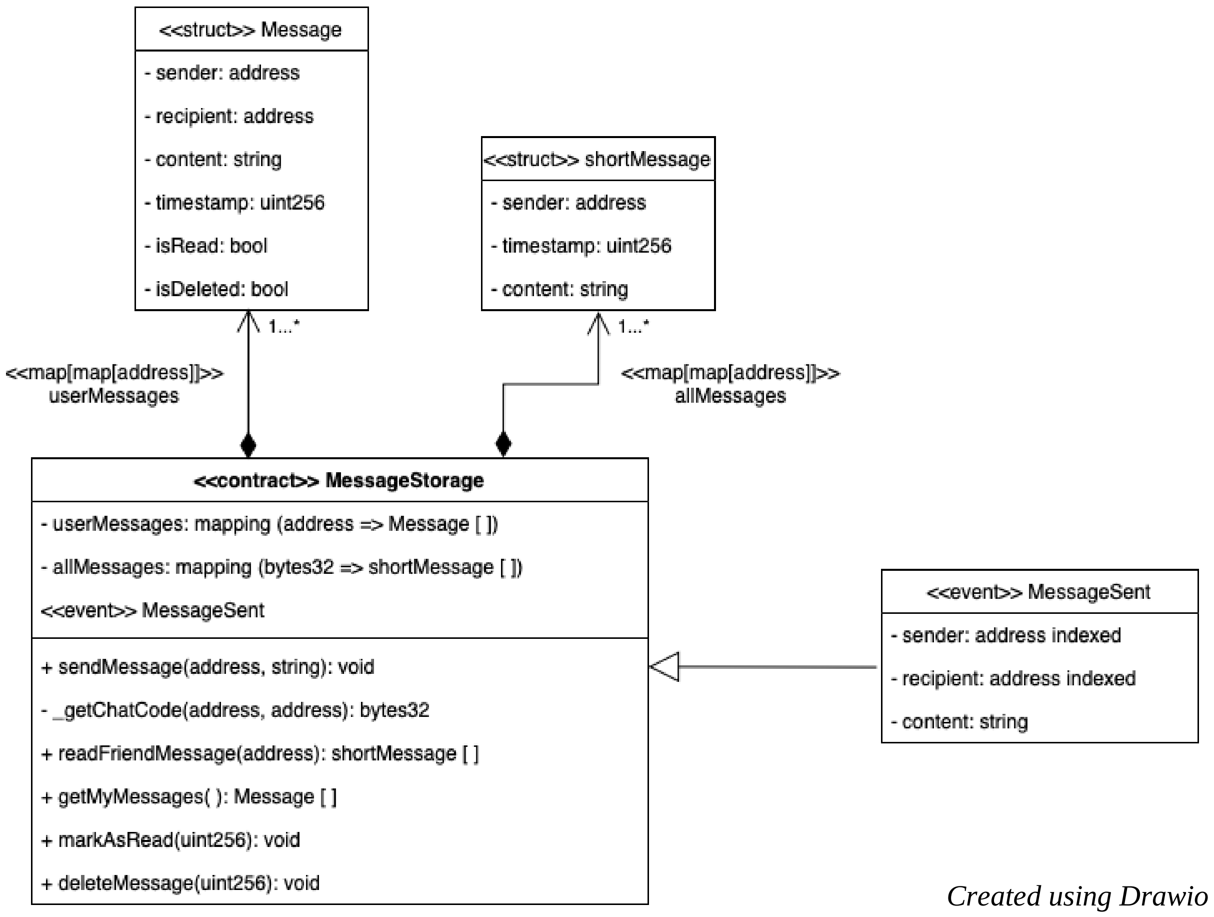
Table 18: User Contract Functions

5.2.2 MESSAGE STORAGE CONTRACT

The message storage contract handles the sending and retrieving of messages between users. When a new message is sent, it will be stored in the Message struct that will be mapped to user's Ethereum addresses for easy retrieval by other functions. The core functionality of the contract revolves around the "sendMessage" function, where users can send messages to each other. For this contract, an event is emitted to notify the frontend user interface when a new message has been sent.

The full *MessageStorage.sol* code can be found in the Github Repository in Appendix A.

The below diagram provides a structural overview of the contract, including the data structures and key variables used.


<<struct>> Message
- sender: address
- recipient: address
- content: string
- timestamp: uint256
- isRead: bool
- isDeleted: bool
<<struct>> shortMessage
- sender: address
- timestamp: uint256
- content: string
1...*
1...*
<<map[map[address]]>> userMessages
<<map[map[address]]>> allMessages
<<contract>> MessageStorage
- userMessages: mapping (address => Message [ ])
- allMessages: mapping (bytes32 => shortMessage [ ])
<<event>> MessageSent
+ sendMessage(address, string): void
- _getChatCode(address, address): bytes32
+ readFriendMessage(address): shortMessage [ ]
+ getMyMessages( ): Message [ ]
+ markAsRead(uint256): void
+ deleteMessage(uint256): void
<<event>> MessageSent
- sender: address indexed
- recipient: address indexed
- content: string

Created using Drawio

Figure 11: Message Storage Contract Class Diagram

The following includes a more detailed explanation of the structs and variables implemented in the MessageStorage smart contract.

Variable Name	Variable Type	Description
Message	Struct	The data of message sent, including user address, recipient addess, message content, timestamp, and two Boolean flags of whether the message isRead and whether it has been deleted.
shortMessage	Struct	The data of message sent, only includes sender address, timestamp and message content.
userMessage	mapping(address => Message[ ])	User's Ethereum address is a unique key that corresponds to the struct which contains the data of all the messages the user has. Mapping enables quick and efficient retrieval of message based on their addresses.
allMessages	mapping(bytes32 => shortMessage[ ])	This mapping associates each chat code (computed based on user's addresses from _getChatCode function) with an array of shortMessage structs. This array stores a simplified representation of messages for easy retrieval by other functions.

Table 19: Message Storage Contract Variables

The contract contains the following functions:

Function	Description
sendMessage	Takes in the address of the recipient of the message and the string of message content to send a message. A new message entry, *newMessage*, will be created and pushed into the sender's *userMessages* array and vice versa for the reciever's *userMessages* array. Calls the *_getChatCode* function, passing in sender's address and recipient's address as parameters to obtain the bytes32 *chatCode*. A new message entry, newshortMessage, will be created and pushed into the *allMessages* array associated with *chatCode*. Emits an event *MessageSent* when function is successful.
_getChatCode	Takes in two user addresses as parameters and returns the keccak256 hash of the concatenation of the two addresses to generate a unique identifier for a chat between two users.
readFriendMessage	Takes in the address of the friend as input parameter. Calls the *_getChatCode* function, to calculate two different chat codes. One is based on current user and provided friend's address, and the other based on swapping the order of the addresses to ensure that the chat codes are unique. Retrieve the arrays of *shortMessage* associated with the two chat codes from the *allMessages* mapping. The variable *totalLength* stores the total length of messages. Initialise a new dynamic array, *combinedMessages*, of *shortMessage* struct with length of *totalLength* to store combined message of both users. Copy messages of first and second user into the *combinedMessages* array using for loops. Returns combined array of messages.
getMyMessages	Returns all the messages of the user from the *userMessages* array.
markAsRead	Takes in a *uint256* as input parameter. Ensures that message exist in the *userMessages* array.

	Alter the Boolean value, *isRead*, of the message from the *userMessages* array to true.
deleteMessage	Takes in a *uint256* as input parameter. Ensures that message exist in the *userMessages* array. Alter the Boolean value, *isDeleted*, of the message from the *userMessages* array to true.

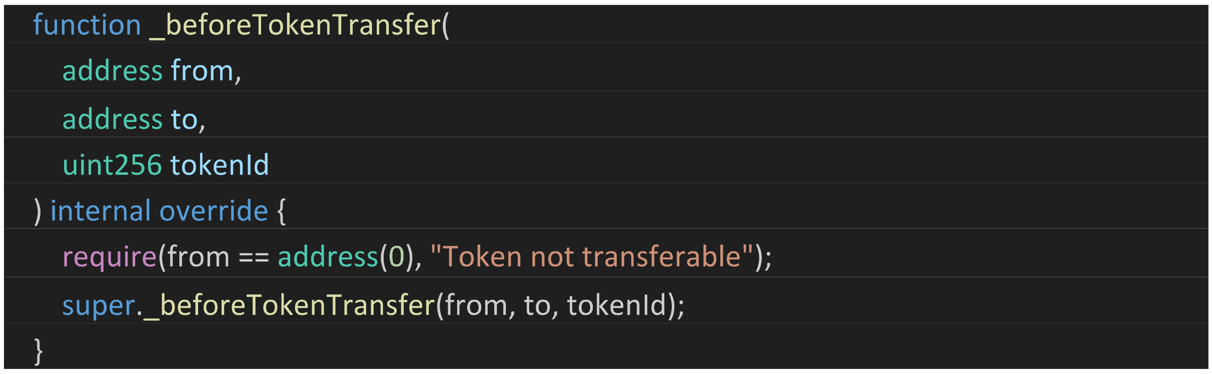
Table 20: Message Storage Contract Functions

5.2.3 SOULBOUND CONTRACT

The Soulbound contract handles the minting of a new SBT. To turn a non-fungible token (NFT) into SBT, the non-transferability function is implemented. The *Soulbound.sol* contract developed inherits the *ERC721.sol* contract from the OpenZeppelin library but overrides the transfer functionality. The overridden transfer function will not transfer and will be reverted with the message “Token not transferrable”.

The full *Soulbound.sol* code can be found in the Github Repository in Appendix A.

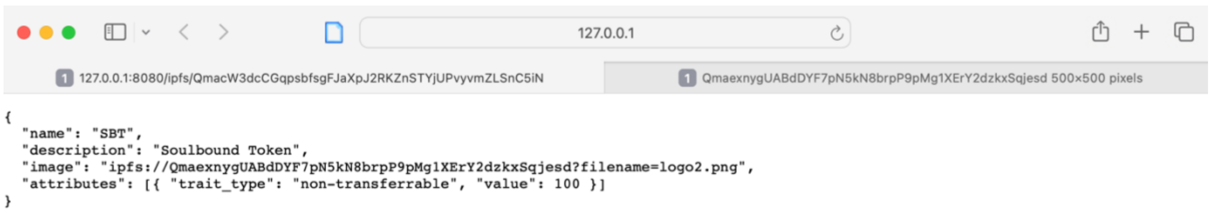
The following shows the _beforeTokenTransfer function of the smart contract that was overridden.



```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 tokenId
) internal override {
    require(from == address(0), "Token not transferable");
    super._beforeTokenTransfer(from, to, tokenId);
}
```

Figure 12: Code for _beforeTokenTransfer Function

For development purposes, a default tokenURI hosted on IPFS is created: ipfs://QmacW3dcCGqpsbfsgFJaXpJ2RKZnSTYjUPvyvmZLSnC5iN


127.0.0.1
127.0.0.1:8080/ipfs/QmacW3dcCGqpsbfsgFJaXpJ2RKZnSTYjUPvyvmZLSnC5iN
QmaexnygUABdDYF7pN5kN8brpP9pMg1XErY2dzkxSqjesd 500×500 pixels

```
{
  "name": "SBT",
  "description": "Soulbound Token",
  "image": "ipfs://QmaexnygUABdDYF7pN5kN8brpP9pMg1XErY2dzkxSqjesd?filename=logo2.png",
  "attributes": [{ "trait_type": "non-transferrable", "value": 100 }]
}
```

Figure 13: Example of tokenURI

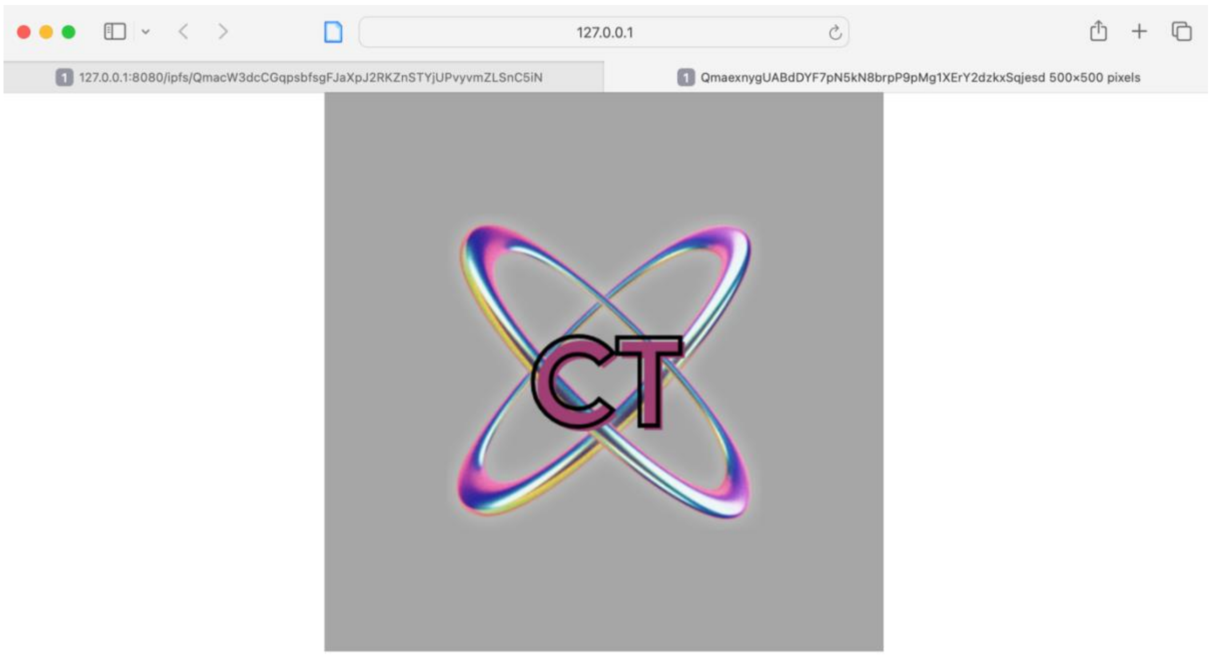

127.0.0.1
127.0.0.1:8080/ipfs/QmacW3dcCGqpsbfsgFJaXpJ2RKZnSTYjUPvyvmZLSnC5iN
QmaexnygUABdDYF7pN5kN8brpP9pMg1XErY2dzkxSqjesd 500×500 pixels
CT

Figure 14: Example of SBT image retrieved from Token URI

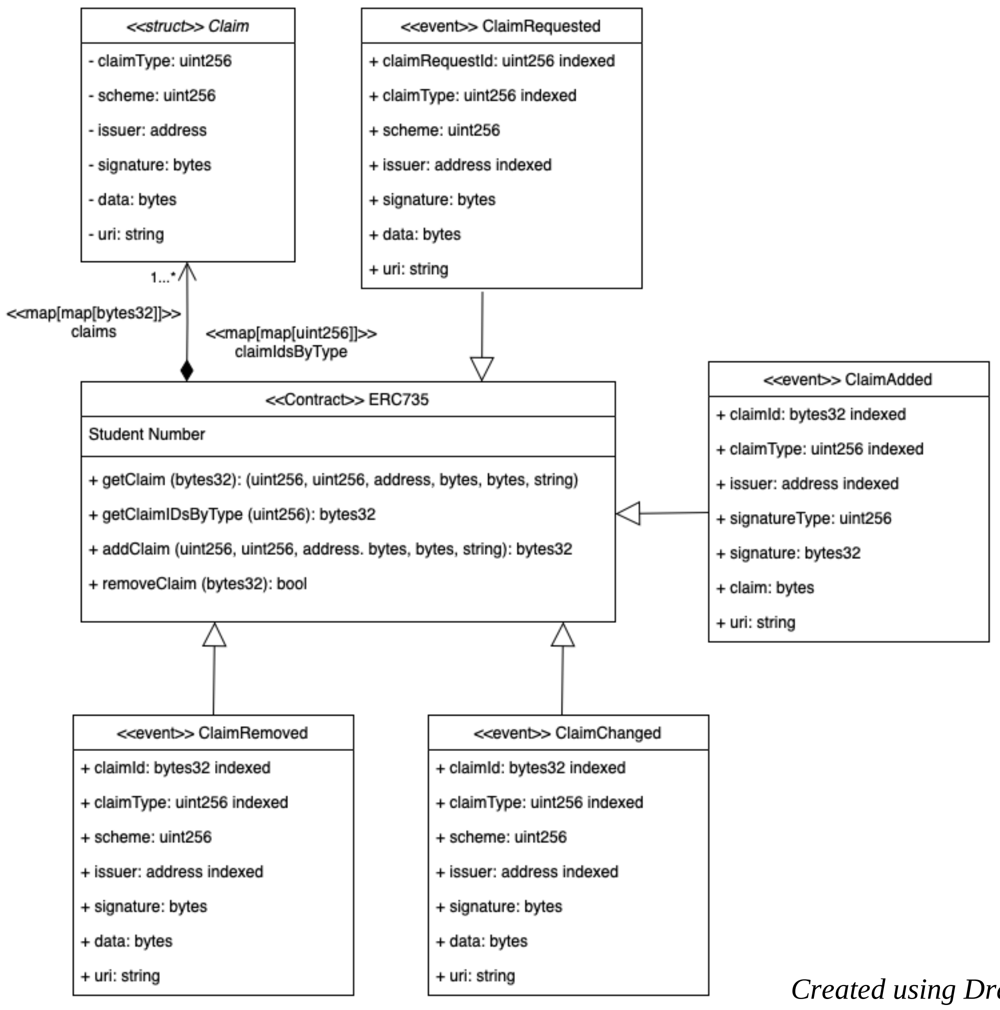
5.2.4 ERC735 Contract

The *ERC735.sol* is developed using the ERC-735 contract standard to allow Dapps and smart contracts to add, remove and hold claims.

The ERC735 contract standard is developed by Fabian Vogelsteller retrieved from: https://github.com/ethereum/eips/issues/735

The full *ERC735.sol* code can be found in the Github Repository in Appendix A.

The below diagram provides a structural overview of the contract, including the data structures and key variables used.



<<struct>> Claim
- claimType: uint256
- scheme: uint256
- issuer: address
- signature: bytes
- data: bytes
- uri: string
<<event>> ClaimRequested
+ claimRequestId: uint256 indexed
+ claimType: uint256 indexed
+ scheme: uint256
+ issuer: address indexed
+ signature: bytes
+ data: bytes
+ uri: string
1...*
<<map[map[bytes32]]>> claims
<<map[map[uint256]]>> claimIdsByType
<<Contract>> ERC735
Student Number
+ getClaim (bytes32): (uint256, uint256, address, bytes, bytes, string)
+ getClaimIDsByType (uint256): bytes32
+ addClaim (uint256, uint256, address. bytes, bytes, string): bytes32
+ removeClaim (bytes32): bool
<<event>> ClaimAdded
+ claimId: bytes32 indexed
+ claimType: uint256 indexed
+ issuer: address indexed
+ signatureType: uint256
+ signature: bytes32
+ claim: bytes
+ uri: string
<<event>> ClaimRemoved
+ claimId: bytes32 indexed
+ claimType: uint256 indexed
+ scheme: uint256
+ issuer: address indexed
+ signature: bytes
+ data: bytes
+ uri: string
<<event>> ClaimChanged
+ claimId: bytes32 indexed
+ claimType: uint256 indexed
+ scheme: uint256
+ issuer: address indexed
+ signature: bytes
+ data: bytes
+ uri: string

Created using Drawio

Figure 15: ERC735 Contract Class Diagram

The claim structure consist of 6 fields. These claims are issues to the identity.

```
struct Claim {
    uint256 claimType;
    uint256 scheme;
    address issuer; // msg.sender
    bytes signature; // this.address + topic + data
    bytes data;
    string uri;
}
```

Figure 16: Code for Claim Struct

The contract contains the following functions:

Function	Description
getClaim	Takes in _claimId as input. Retrieves the claim struct from the *claims* mapping. Returns claimaType, scheme, issuer, signature,, data and uri of the claim.
getClaimIdsByType	Takes in _claimType as input. Retrieves the ClaimID from *claimIdsByType* mapping. Returns an array of claim IDs by type.
addClaim	Takes in _clainType, _scheme, _issuer, _signature, _data and _uri as input. Creates a new claim entry and add it into the *claims* array. Push the new claim ID into the *claimIdsByType* mapping. Emit an event for the successful creation of the claim.
removeClaim	Takes in _claimId as input. Ensure that the user calling this function is the owner of the claim. Remove the claim from the *claims* mapping. Update the index of the claim IDs by type to reflect actual removal of claim. Emit an event for the successful removal of the claim. Returns a Boolean value true after function is executed.

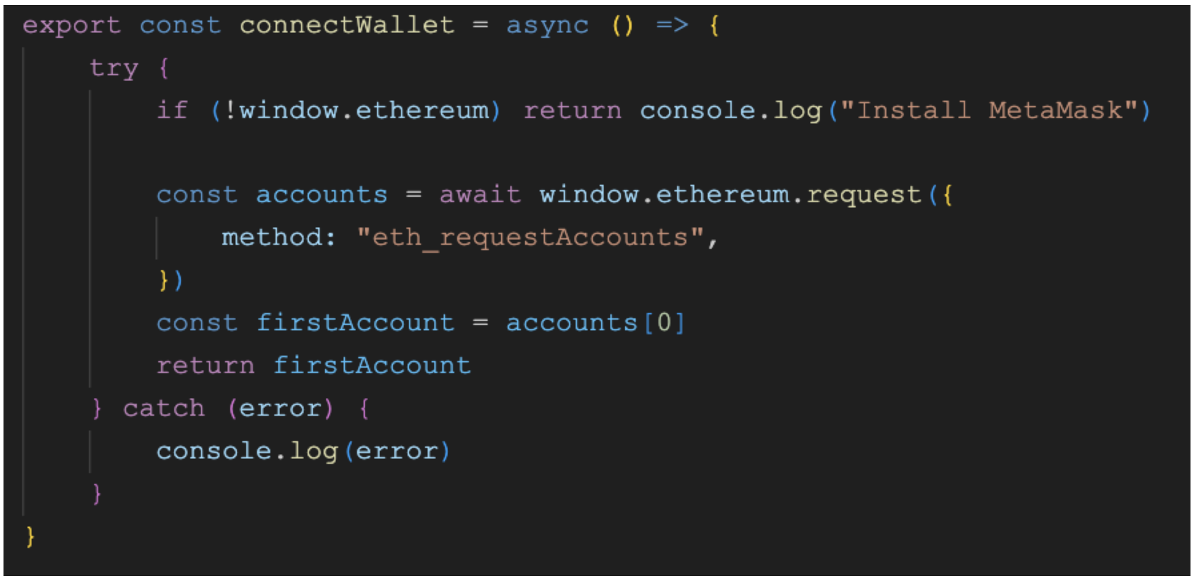
Table 21: ERC735 Contract Functions

5.3 Front-end Development

The frontend web application is developed using the ReactJS Web framework. The web interface is a fully functional messaging platform for Users to exchange messages without having to interact directly with the command line. Metamask, ethers.js and Web3Modal is used to handle user's wallet connection and interacting with the smart contracts.

5.3.1 Metamask Wallet Connection

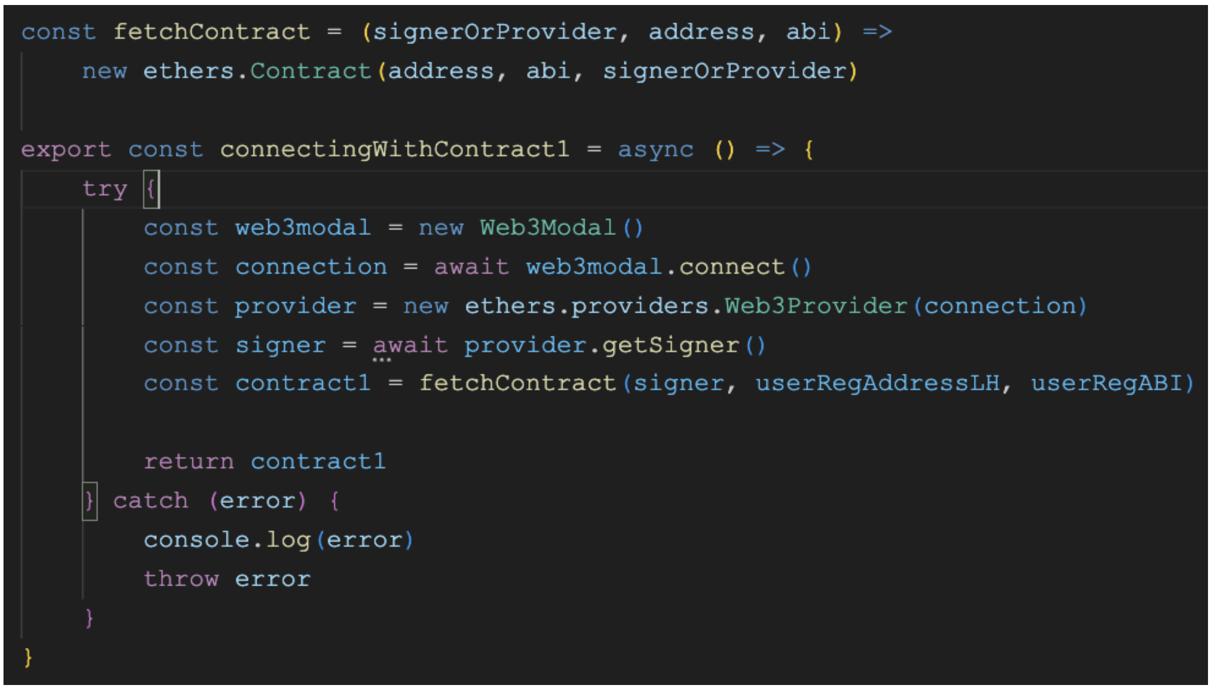
The below code is designed to connect to the user's Ethereum wallet using MetaMask. It first checks if the *window.ethereum* object exist to check if the MetaMask extension is installed. If MetaMask is installed, the function uses the *window.ethereum.request* method to request the Ethereum accounts connected to MetaMask. The *eth_requestAccounts* method is used to request access to user's accounts. It then retrieves the first Ethereum account from the list of accounts obtained and returns it. This will be the address of the connected account.



```
export const connectWallet = async () => {
    try {
        if (!window.ethereum) return console.log("Install MetaMask")

        const accounts = await window.ethereum.request({
            method: "eth_requestAccounts",
        })
        const firstAccount = accounts[0]
        return firstAccount
    } catch (error) {
        console.log(error)
    }
}
```

Figure 17: Code for Connecting Wallet

The below code is designed to interact with an Ethereum smart contract using the ether.js library. A utility function, *fetchContract* that takes a signer or provider, a contract address and the contract ABI is defined. It creates and returns a new instance of an ethers.js *Contract* using the provided parameters. The *connectingWithContract1* function first creates an instance of Web3Modal for handling the connection to user's wallet. The *connect* methods of the Web3Modal instance is used to connect to the user's wallet. An ethers.js Web3Provider Is created using the connected wallet to interact with the Ethereum blockchain. The signer (account) from the connected wallet is retrieved using the *getSigner* method. The *fetchContract* utility function is called to create a new *Contract* instance using the obtained signer, address and ABI. The instantiated smart contract, *contract1*, is returned to the caller which can be used to interact with the deployed smart contract on the Ethereum blockchain.



```
const fetchContract = (signerOrProvider, address, abi) =>
    new ethers.Contract(address, abi, signerOrProvider)

export const connectingWithContract1 = async () => {
    try {
        const web3modal = new Web3Modal()
        const connection = await web3modal.connect()
        const provider = new ethers.providers.Web3Provider(connection)
        const signer = await provider.getSigner()
        const contract1 = fetchContract(signer, userRegAddressLH, userRegABI)

        return contract1
    } catch (error) {
        console.log(error)
        throw error
    }
}
```

Figure 18: Code to fetch contract

5.3.2 APPLICATION WALKTHROUGH

This is the first landing page for a new user. The user has to first connect their MetaMask wallet using the browser extension.

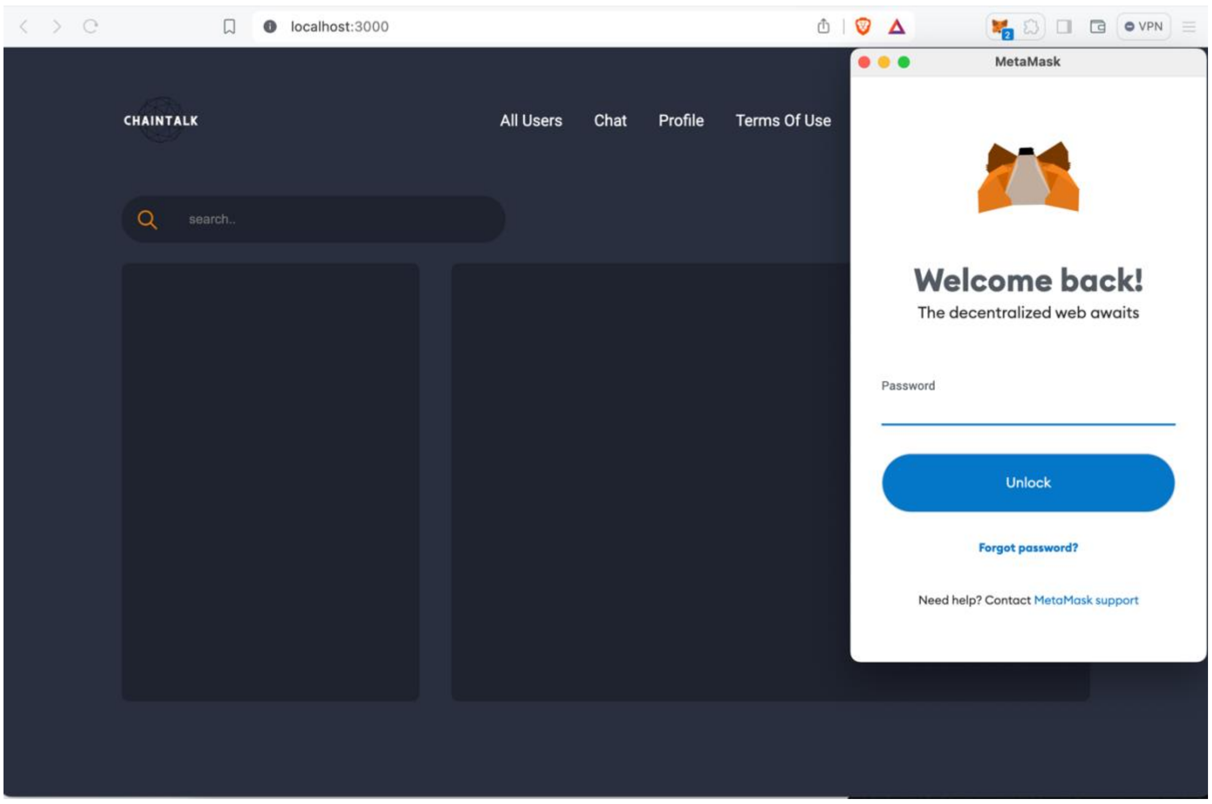
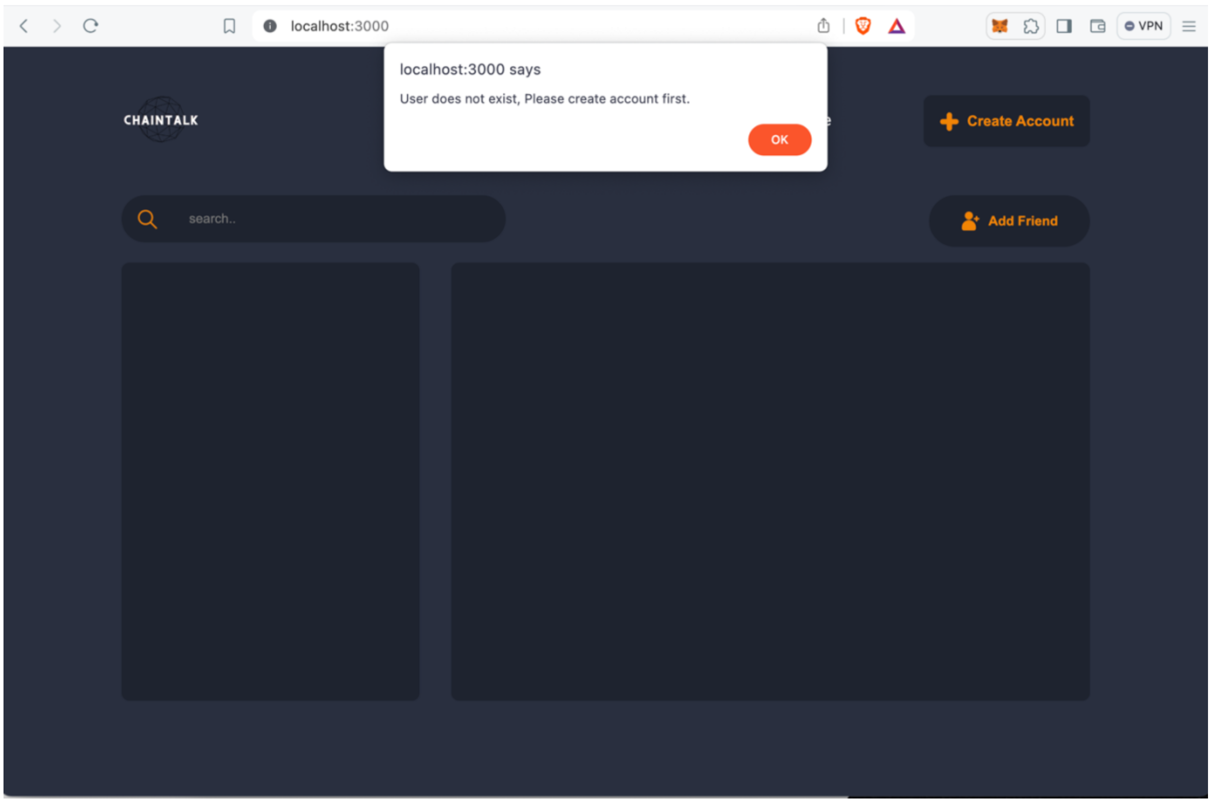


Figure 19: Landing Page for New User

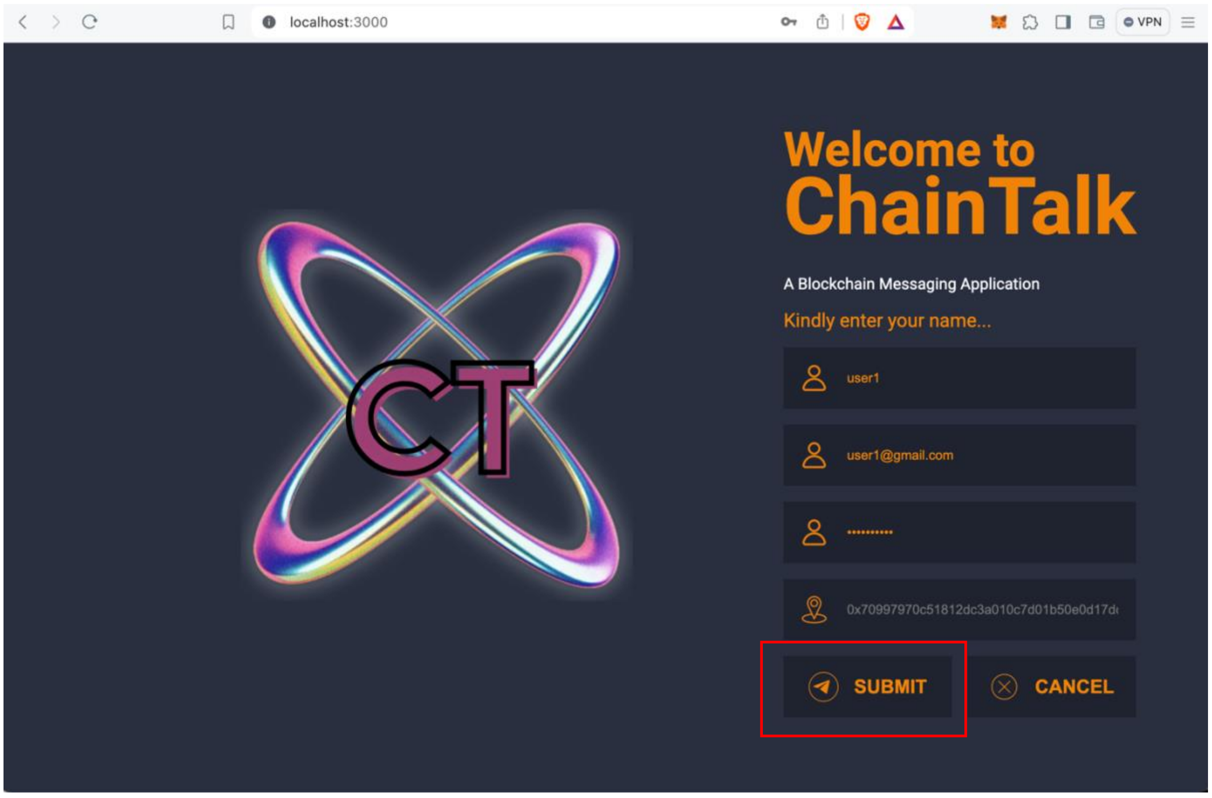
As a new user that does not has an account on ChainTalk, a notification will be displayed on the homepage “User does not exist. Please create an account first.”


localhost:3000 says

User does not exist, Please create account first.

Figure 20: Homepage for New User

Upon clicking on "Create Account" button on the top right hand corner of the webpage, user will be directed to the User Registration page to register for an account. In the user registration page, the new user has to key in their username, email address and their desired password. After which, the user can click on the "Submit" button to create an account.



user1

user1@gmail.com

••••••••••

0x70997970c51812dc3a010c7d01b50e0d17d

Figure 21: User Registration Page

Upon successful creation of an account, the new user will be directed to the All Users Page which is also the homepage, where they will able to see the name and address of all the registered users on the application, including themselves.
A user can also choose to add friends through this page by clicking on the “Add Friend” button, or proceed to the Chat Page to add their friends.
The “Create Account” button on the top right corner of the webpage will also be replaced with the existing User’s username.

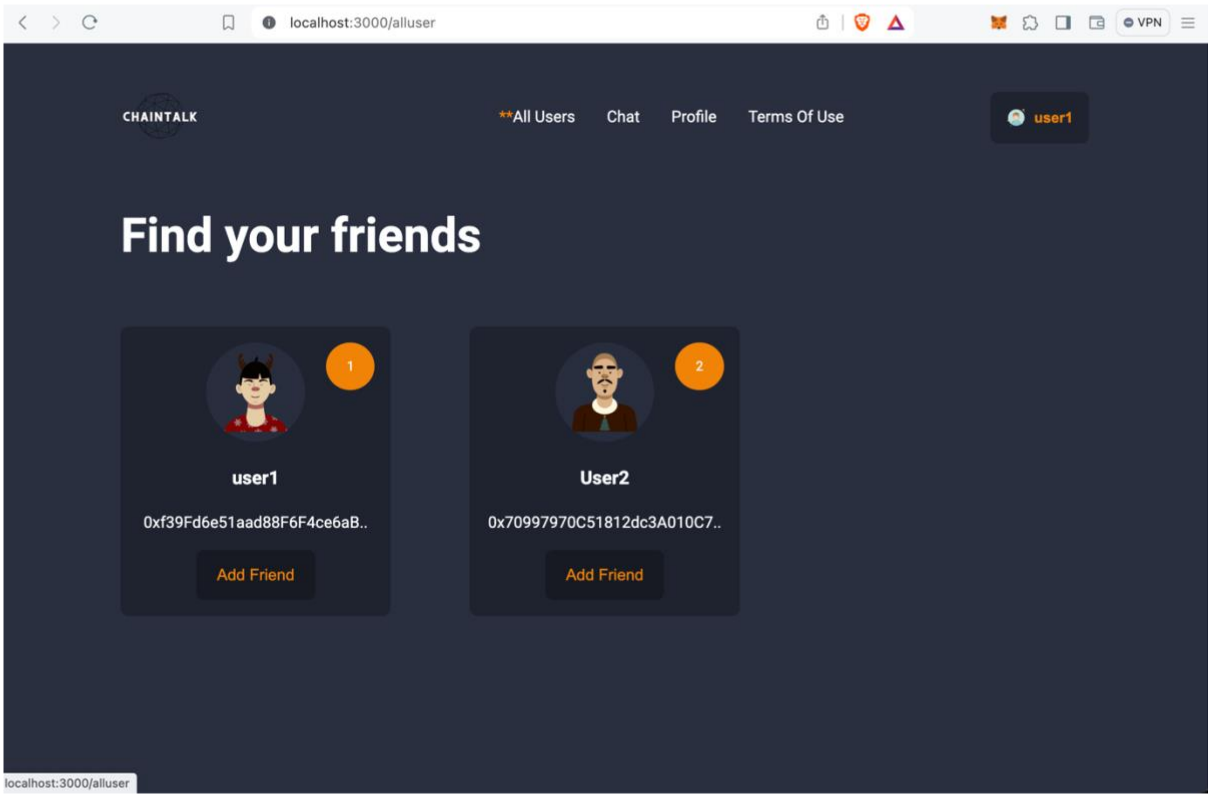


Figure 22: All Users page

If a user has multiple accounts, they can toggle between their accounts by clicking on their username on the top right corner of the webpage to switch between different accounts using the MetaMask extension. Below also reflects the Chat page.

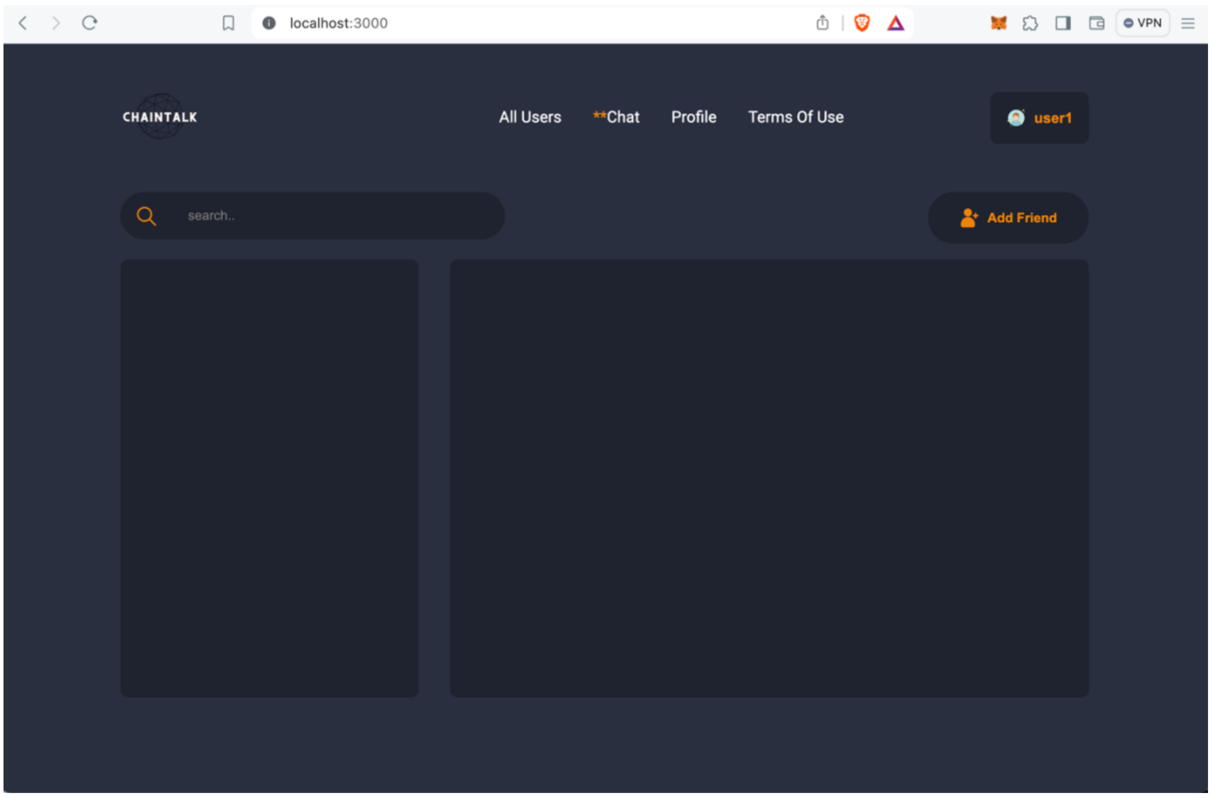


Figure 23: Chat Page for New User

Upon clicking on the "Add Friend" button on the Chat Page, the user will be directed to the Add Friend page. In this page, user can key in the address of the user they wish to add as friend. After which, the user can click on the "Submit" button to add the user as friend.

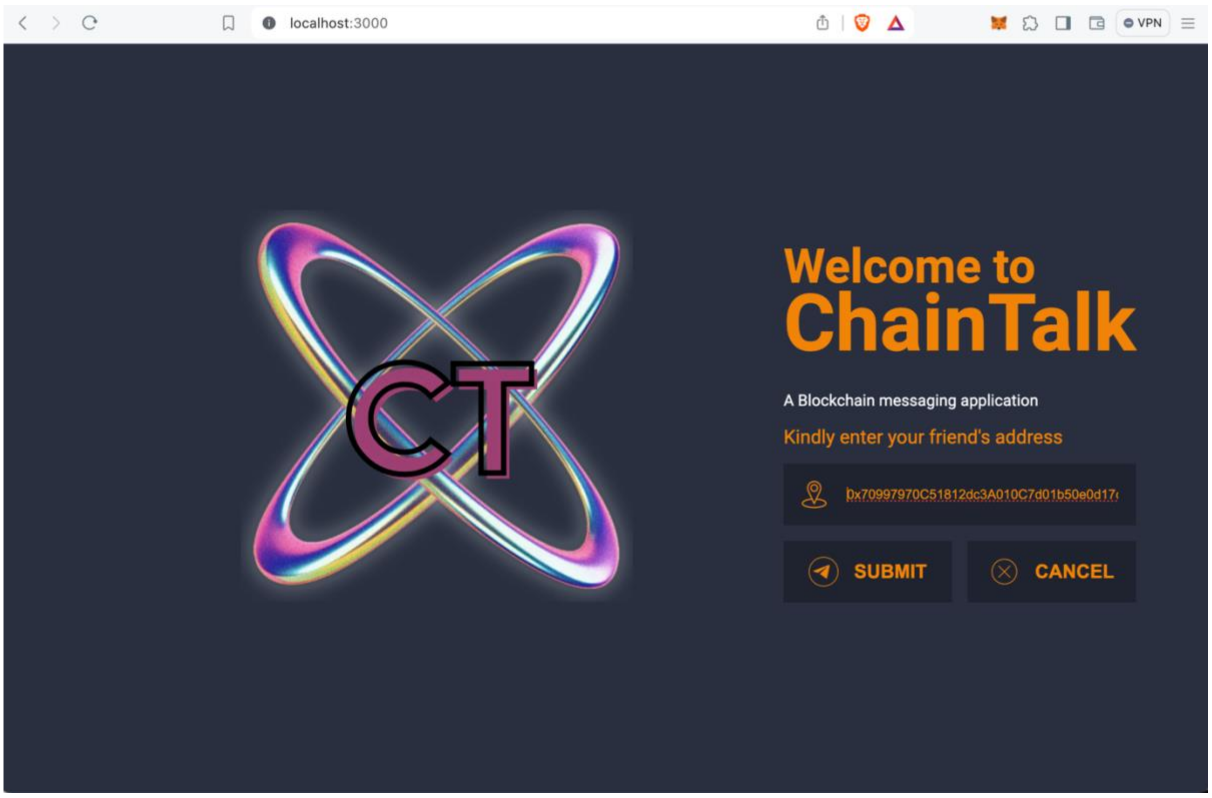
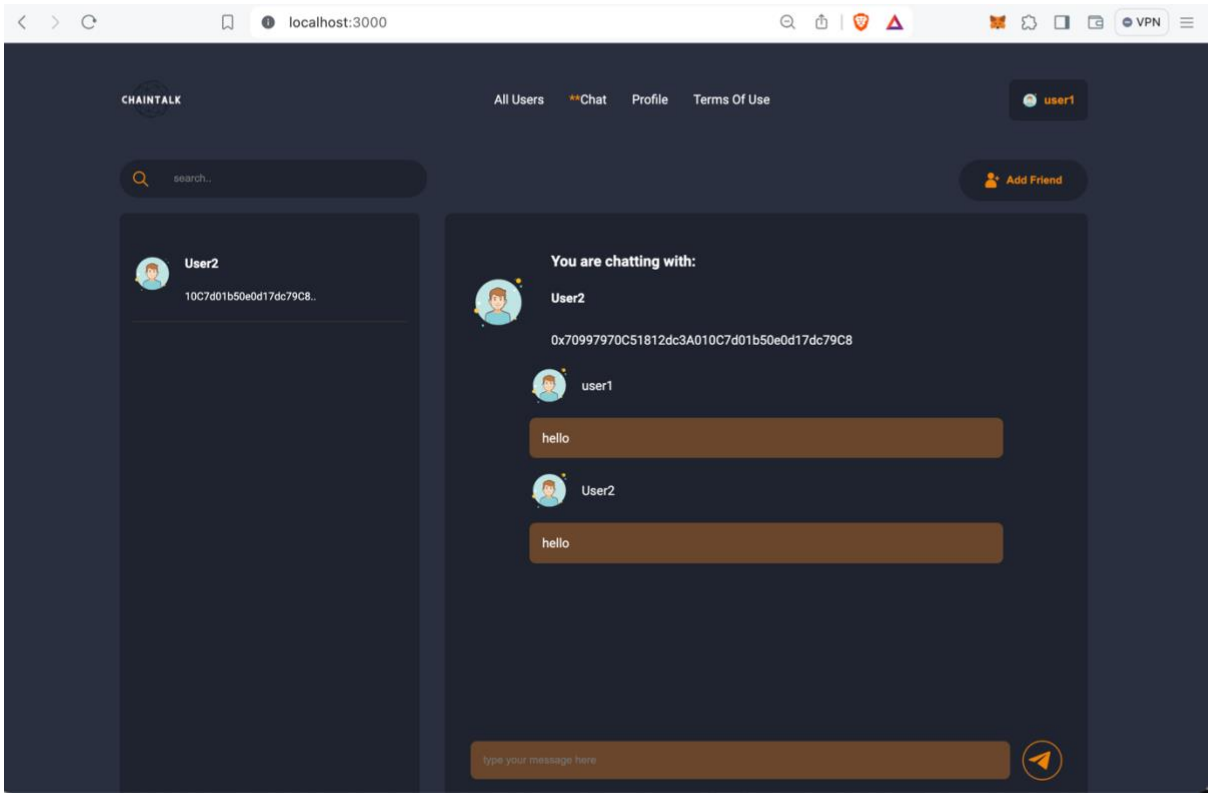

localhost:3000
VPN
Welcome to
ChainTalk
A Blockchain messaging application
Kindly enter your friend's address
0x70997970C51812dc3A010C7d01b50e0d17
SUBMIT
CANCEL

Figure 24: Add Friend Page

After two users are being added as friends, the user can now see their friend's name and address in the left panel in the Chat Page. The user can click on the name of the User they wish to chat with on the left panel, and the chat history will be reflected in the right panel.

The user can send a message to their friend by typing in the message content in the orange input text field and clicking on the submit icon to send the message to their friend. The user they are chatting with is also reflected in the right panel of the page.



User2
10C7d01b50e0d17dc79C8..

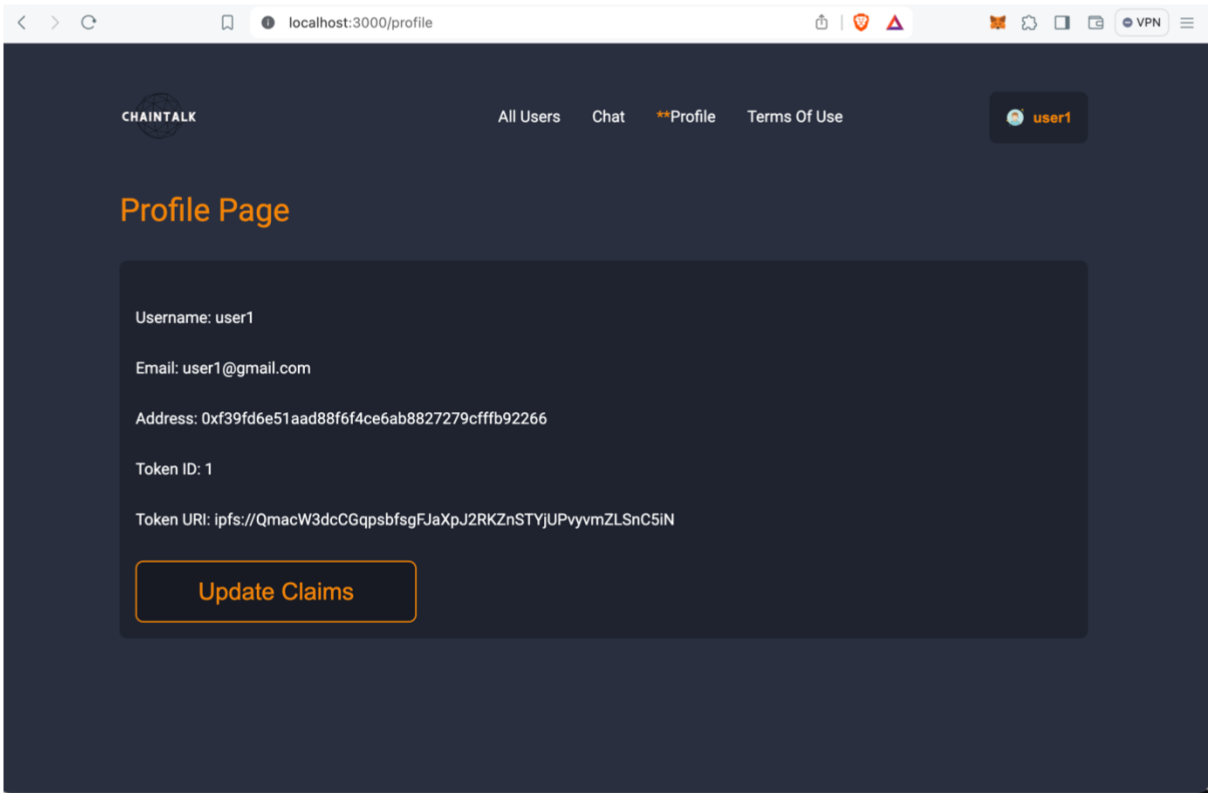
You are chatting with:
User2
0x70997970C51812dc3A010C7d01b50e0d17dc79C8

user1
hello

User2
hello

Figure 25: Chat Page after Adding of Friends

The user will be able to view their Profile by clicking on the “Profile” tab on the navigation bar. The user’s Username, Email, Account Address, Token ID and Token URI will be displayed in the Profile page. Additionally, the user can manually update their claims in the Identity contract by clicking onto the “Update Claims” button.



Profile Page

Username: user1

Email: user1@gmail.com

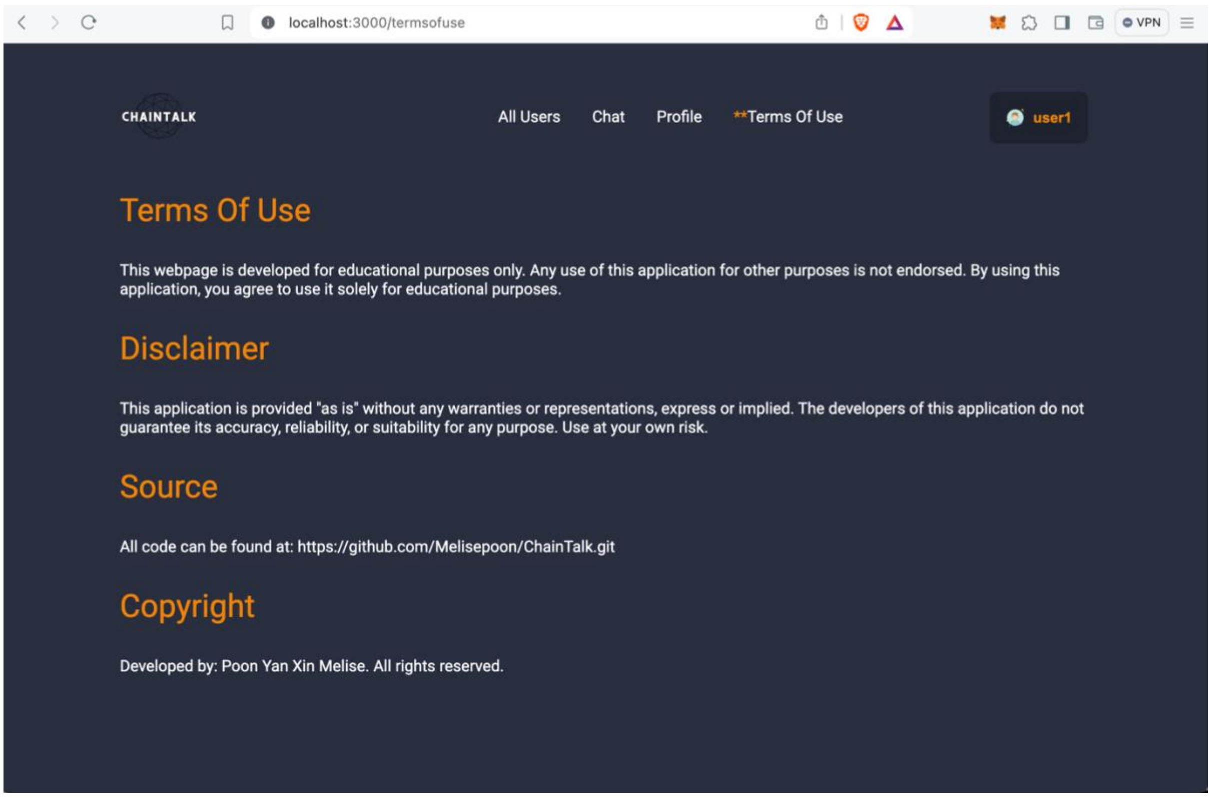
Address: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266

Token ID: 1

Token URI: ipfs://QmacW3dcCGqpsbfsgFJaXpJ2RKZnSTYjUPvyvmZLSnC5iN

Figure 26: User Profile Page

The user can view the Terms of Use page by clicking on the “Terms Of Use” tab on the navigation bar. This page depicts the general Disclaimers and Copyrights of the website.



Terms Of Use

This webpage is developed for educational purposes only. Any use of this application for other purposes is not endorsed. By using this application, you agree to use it solely for educational purposes.

Disclaimer

This application is provided "as is" without any warranties or representations, express or implied. The developers of this application do not guarantee its accuracy, reliability, or suitability for any purpose. Use at your own risk.

Source

All code can be found at: https://github.com/Melisepoon/ChainTalk.git

Copyright

Developed by: Poon Yan Xin Melise. All rights reserved.

Figure 27: Terms Of Use Page

Chapter 6 : TESTING

6.1 SMART CONTRACT TEST SCRIPTS

Test scripts for the Solidity smart contracts are created to ensure correctness, security and reliability of the codebase. The purpose is to identify and preventing bugs and vulnerabilities and facilitating code changes and refactoring. This contributes to an automated testing workflow which plays and integral part in building confidence in the solidity smart contract system.

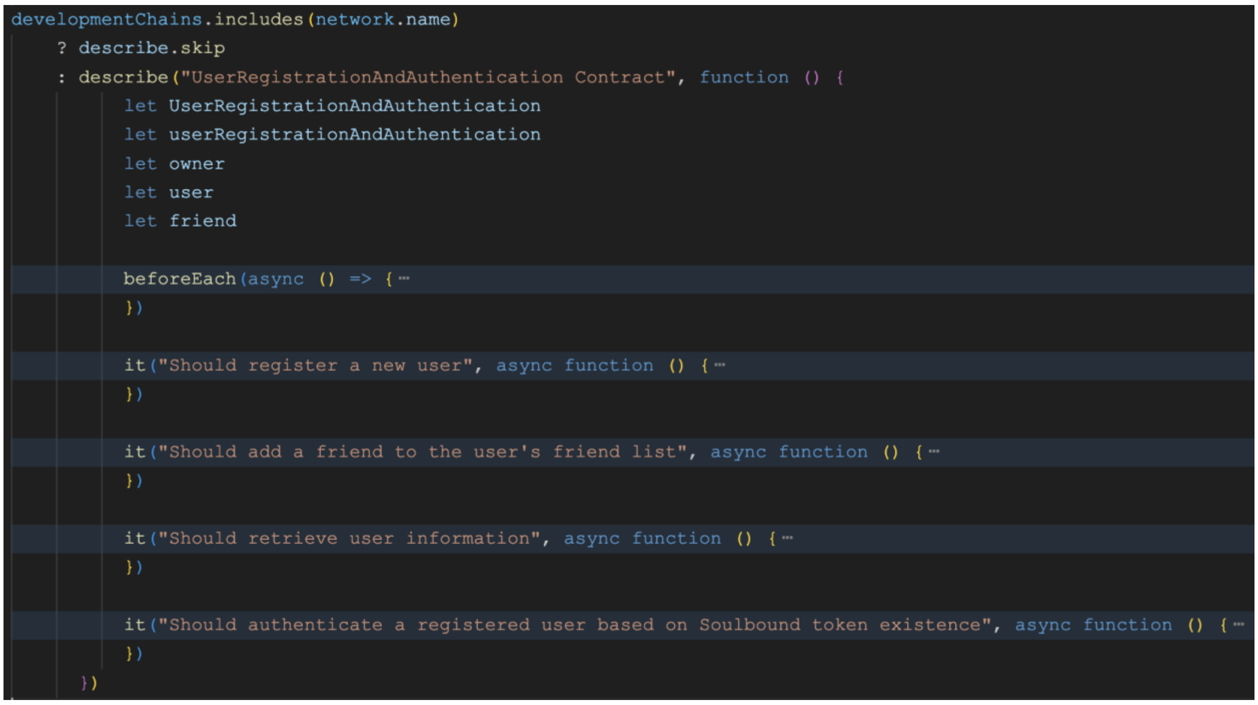
The below JavaScript test scripts implemented uses the Mocha testing framework along with the Chai assertion library to test the contracts.

6.1.1 User Contract Test Scripts

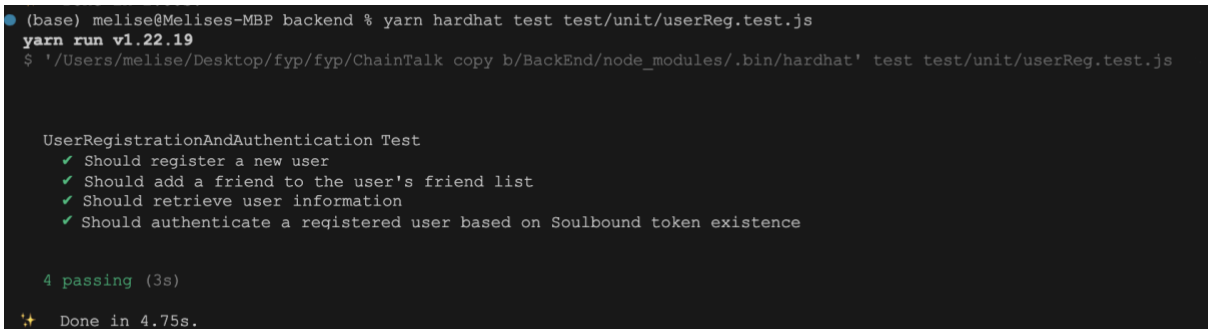
The following summarises the test scripts created for the User contract. The full code for the test scripts can be found in the Github Repository in Appendix A.

The test cases the script checks for includes registration of a user, retrieving user information, adding a friend and authenticating a user.

The script first checks if the network is a development chain. If it is, the entire test suite is skipped. This is meant to exclude the test from running on Testnets or other non-development environment. The *beforeEach* function is used to set up the environment before each individual test by deploying the *UserRegistrationAndAuthentication* contract. Two accounts are created, one deployer and one user for testing different user interactions in the system. The first test case includes the script that focuses on registering a user and retrieving their information by using the *registerUser* function and *getMyUserInfo* function of the deployed contract.


```
developmentChains.includes(network.name)
    ? describe.skip
    : describe("UserRegistrationAndAuthentication Contract", function () {
          let UserRegistrationAndAuthentication
          let userRegistrationAndAuthentication
          let owner
          let user
          let friend

          beforeEach(async () => {⋯
          })

          it("Should register a new user", async function () {⋯
          })

          it("Should add a friend to the user's friend list", async function () {⋯
          })

          it("Should retrieve user information", async function () {⋯
          })

          it("Should authenticate a registered user based on Soulbound token existence", async function () {⋯
          })
      })
```
Figure 28: Condensed test Scripts for User Contract

The below image shows the running of the User contract test scripts in the terminal using `hardhat test`. The results shows all tests have passed.



```
● (base) melise@Melises-MBP backend % yarn hardhat test test/unit/userReg.test.js
  yarn run v1.22.19
  $ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' test test/unit/userReg.test.js



    UserRegistrationAndAuthentication Test
      ✔ Should register a new user
      ✔ Should add a friend to the user's friend list
      ✔ Should retrieve user information
      ✔ Should authenticate a registered user based on Soulbound token existence


    4 passing (3s)

  ✨  Done in 4.75s.
```

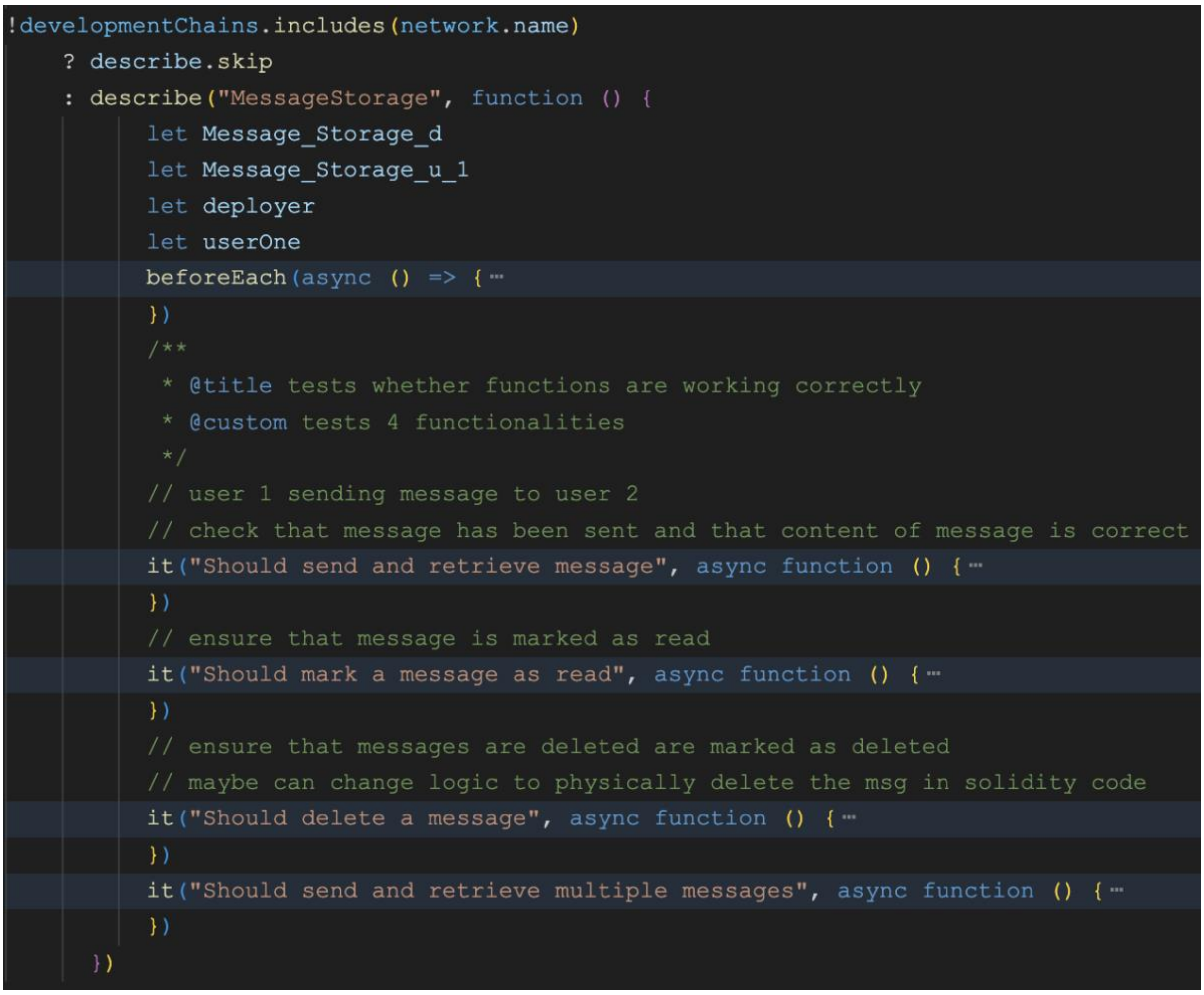
Figure 29: User Contract Test Scripts in the Terminal

6.1.2 MESSAGE CONTRACT TEST SCRIPTS

The following summarises the test scripts created for the Message contract. The full code for the test scripts can be found in the Github Repository in Appendix A.

The test cases the script checks for includes sending and retrieving messages, marking a message as read, deleting a message, and sending and receiving multiple messages.

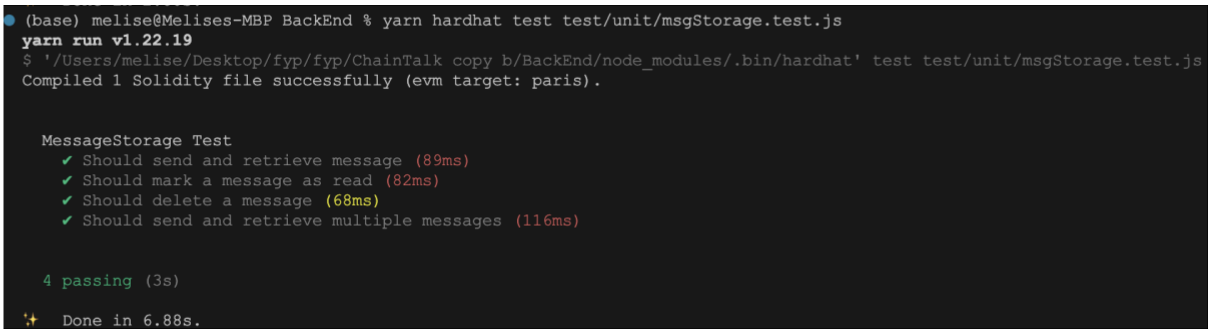
The script first checks if the network is a development chain. If it is, the entire test suite is skipped. This is meant to exclude the test from running on Testnets or other non-development environment. The *beforeEach* function is used to set up the environment before each individual test by deploying the *MessageStorage* contract. Two accounts are created, one deployer and one user for testing different user interactions in the system. The script includes several test cases within the `it` blocks, each focusing on a specific functionality.



```
!developmentChains.includes(network.name)
    ? describe.skip
    : describe("MessageStorage", function () {
          let Message_Storage_d
          let Message_Storage_u_1
          let deployer
          let userOne
          beforeEach(async () => {⋯
          })
          /**
           * @title tests whether functions are working correctly
           * @custom tests 4 functionalities
           */
          // user 1 sending message to user 2
          // check that message has been sent and that content of message is correct
          it("Should send and retrieve message", async function () {⋯
          })
          // ensure that message is marked as read
          it("Should mark a message as read", async function () {⋯
          })
          // ensure that messages are deleted are marked as deleted
          // maybe can change logic to physically delete the msg in solidity code
          it("Should delete a message", async function () {⋯
          })
          it("Should send and retrieve multiple messages", async function () {⋯
          })
      })
```

Figure 30: Condensed test scripts for Message Contract

The below image shows the running of the MessageStorage contract test script in the terminal using `hardhat test`. The results shows all tests have passed.



```
(base) melise@Melises-MBP BackEnd % yarn hardhat test test/unit/msgStorage.test.js
yarn run v1.22.19
$ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' test test/unit/msgStorage.test.js
Compiled 1 Solidity file successfully (evm target: paris).


  MessageStorage Test
    ✔ Should send and retrieve message (89ms)
    ✔ Should mark a message as read (82ms)
    ✔ Should delete a message (68ms)
    ✔ Should send and retrieve multiple messages (116ms)


  4 passing (3s)

✨  Done in 6.88s.
```

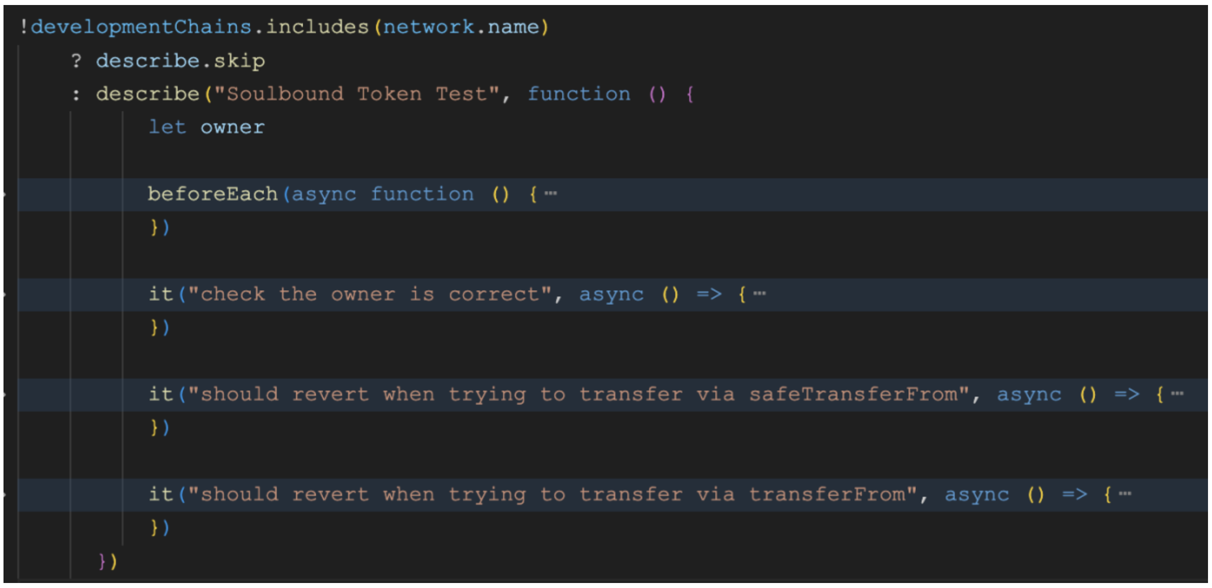
Figure 31: Message Contract Test Scripts in the Terminal

6.1.3 Soulbound Contract Test Scripts

The following summarises the test scripts created for the Soulbound contract. The full code for the test scripts can be found in the Github Repository in Appendix A.

The test cases the script checks for whether the owner of the token is correct, the function should revert if user is trying to transfer the token using both safeTransferFrom function and transferFrom function to ensure that the SBT is non-transferrable.

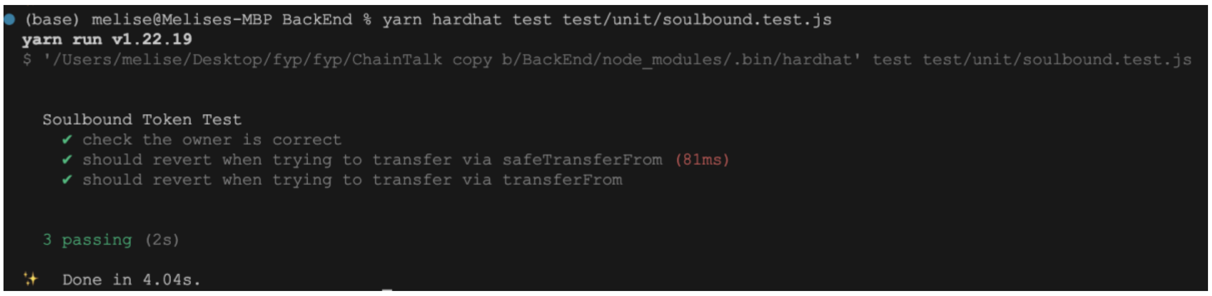
The script first checks if the network is a development chain. If it is, the entire test suite is skipped. This is meant to exclude the test from running on Testnets or other non-development environment. The *beforeEach* function is used to set up the environment before each individual test by deploying the *Soulbound* contract. One account is created as the owner and the Soulbound contract is deployed locally. Subsequently, a SBT is minted by the owner using the safeMint function. The script includes several test cases within the `it` blocks, each focusing on a specific functionality.



```
!developmentChains.includes(network.name)
    ? describe.skip
    : describe("Soulbound Token Test", function () {
          let owner

          beforeEach(async function () {⋯
          })

          it("check the owner is correct", async () => {⋯
          })

          it("should revert when trying to transfer via safeTransferFrom", async () => {⋯
          })

          it("should revert when trying to transfer via transferFrom", async () => {⋯
          })
      })
```

Figure 32: Condensed test scripts for Soulbound Contract

The below image shows the running of the Soulbound contract test scripts in the terminal using `hardhat test`. The results shows all tests have passed.



```
(base) melise@Melises-MBP BackEnd % yarn hardhat test test/unit/soulbound.test.js
yarn run v1.22.19
$ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' test test/unit/soulbound.test.js


  Soulbound Token Test
    ✔ check the owner is correct
    ✔ should revert when trying to transfer via safeTransferFrom (81ms)
    ✔ should revert when trying to transfer via transferFrom


  3 passing (2s)

✨  Done in 4.04s.
```

Figure 33: Soulbound Contract Test Scripts in the Terminal

6.2 BLACK BOX TESTING

Black box testing is a software testing technique where the internal workings or structure of the system under test are not known or considered by the tester. In this approach, the tester focuses solely on the inputs and outputs of the software application, treating it as a "blackbox" whose internal logic is hidden and irrelevant for the testing process.

The below section displays the results after conducting black box testing on the different functionalities of ChainTalk.

6.2.1 USER REGISTRATION

This method take in the following parameters:

- Username
- Email
- Password

Equivalent Classes

Input	Valid	Invalid
Username	Username must be unique across the system	Username already exist in the system.
Email	Email must be unique across the system	Email already exist in the system.
Password	Password must satisfy the following conditions: • At least 8 characters • Have at least 1 alphabet and a number. • Unique across the system	Password that does not satisfy the conditions.

Generic Testing

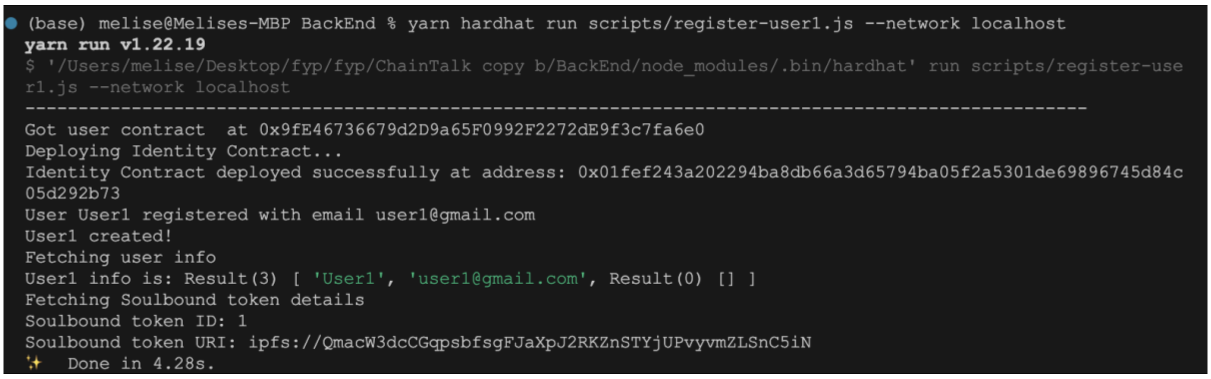
Test ID	Scenario	Expected Result	Actual Result
1	Register with a valid username, email and password	The account is created.	The account is created.
2	Register with an invalid username that has incorrect format.	The system will display and error message and clear the field.	The system will display and error message and clear the field.
3	Register with an invalid email that has incorrect format.	The system will display and error message and clear the field.	The system will display and error message and clear the field.
4	Register with an invalid password that has incorrect format.	The system will display and error message and clear the field.	The system will display and error message and clear the field.

Specific Cases

Test ID	Username	Email	Password	Expected Result	Actual Result
1	User1	user1@gmail.com	Password123	The account is created.	The account is created.
2	User2	User2@gmail.com	Password123	The account is created.	The account is created.

Test Results using Backend Scripts

The figure below shows the registration process of User1. The data of the registered user is also printed out to ensure the successful creation of the account.



```
● (base) melise@Melises-MBP BackEnd % yarn hardhat run scripts/register-user1.js --network localhost
  yarn run v1.22.19
  $ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' run scripts/register-use
  r1.js --network localhost
  ----------------------------------------------------------------------------------------------------
  Got user contract  at 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0
  Deploying Identity Contract...
  Identity Contract deployed successfully at address: 0x01fef243a202294ba8db66a3d65794ba05f2a5301de69896745d84c
  05d292b73
  User User1 registered with email user1@gmail.com
  User1 created!
  Fetching user info
  User1 info is: Result(3) [ 'User1', 'user1@gmail.com', Result(0) [] ]
  Fetching Soulbound token details
  Soulbound token ID: 1
  Soulbound token URI: ipfs://QmacW3dcCGqpsbfsgFJaXpJ2RKZnSTYjUPvyvmZLSnC5iN
  ✨  Done in 4.28s.
```

Figure 34: Registration of User1 using Scripts

6.2.2 ADD FRIEND

This method takes in the following parameters:

- Ethereum Account Address.

<u>Equivalent Classes</u>

Input	Valid	Invalid
Ethereum Account Address	Address maps to a user that has an existing account (found in database) Address maps to a user that is already friends with existing user.	An address that does not map to an existing account

<u>Generic Testing</u>

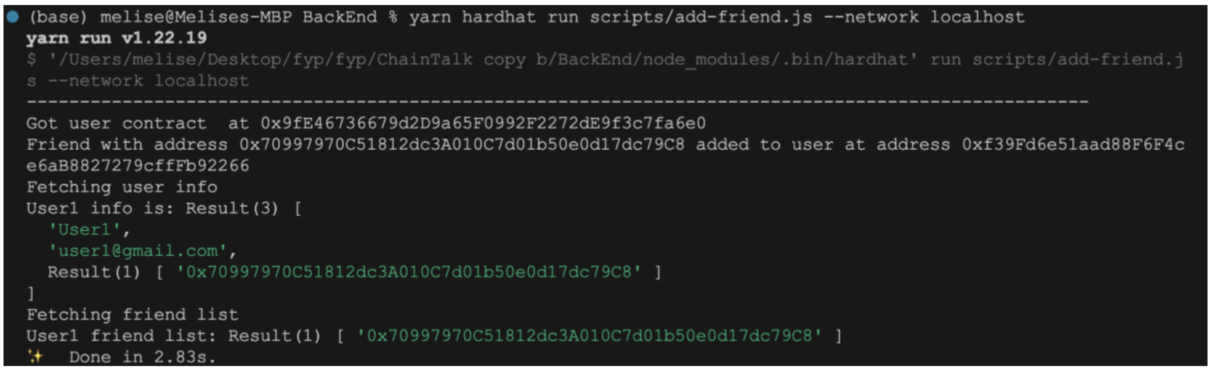
Test ID	Scenario	Expected Result	Actual Result
1	User keys in valid Ethereum account address that maps to a registered user to add as friend.	The two users are added as friends.	The two users are added as friends.
2	User keys in an invalid Ethereum account address that does not belong to a registered user to add as friend.	The system will display and error message “User not registered!”	The system will display and error message “User not registered!”
3	User keys in a valid Ethereum account address that belongs to a registered user that is already friends with current user.	The system will display and error message “Users are already friends!”	The system will display and error message “Users are already friends!”

Specific Cases

Test ID	Current User	Friend’s Ethereum account address	Expected Result	Actual Result
1	User1	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	The users are added as friends.	The users are added as friends.
2	User 1	0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC	Error message “User not registered!” displayed.	Error message “User not registered!” displayed.

Test Results using Backend Scripts

The figure below shows the adding of friends between the registered User1 and User2. The data and friendList of the user is also printed out to ensure the successful adding of friends.



```
● (base) melise@Melises-MBP BackEnd % yarn hardhat run scripts/add-friend.js --network localhost
  yarn run v1.22.19
  $ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' run scripts/add-friend.j
  s --network localhost
  ----------------------------------------------------------------------------------------------------
  Got user contract  at 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0
  Friend with address 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 added to user at address 0xf39Fd6e51aad88F6F4c
  e6aB8827279cffFb92266
  Fetching user info
  User1 info is: Result(3) [
    'User1',
    'user1@gmail.com',
    Result(1) [ '0x70997970C51812dc3A010C7d01b50e0d17dc79C8' ]
  ]
  Fetching friend list
  User1 friend list: Result(1) [ '0x70997970C51812dc3A010C7d01b50e0d17dc79C8' ]
  ✨  Done in 2.83s.
```

Figure 35: Adding of Friends using Scripts

6.2.3 SEND MESSAGE

This method takes in the following parameters:

- Ethereum Account Address
- Message Content

Equivalent Classes

Input	Valid	Invalid
Ethereum Account Address	Address maps to a user that is already in existing user’s friend list. (found in database)	An address that does not map to a user that is in existing user’s friend list
Message Content	String that consist of at least 1 character.	An empty string

Generic Testing

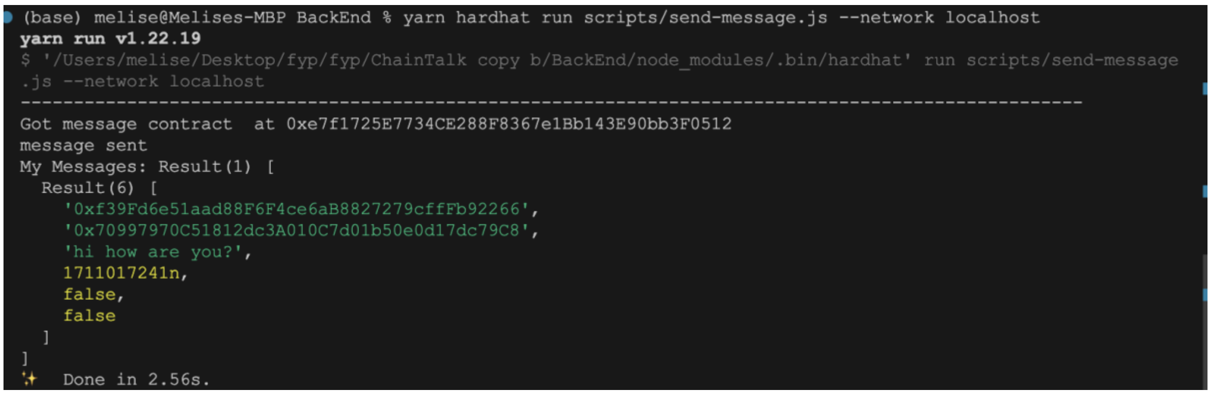
Test ID	Scenario	Expected Result	Actual Result
1	User selects the friend to send a message to in the front-end and enters message content in input field. User clicks on “Send” icon to send the message.	The message is successfully sent.	The message is successfully sent.
2	User selects the friend to send a message to in the front-end and does not enter message content in input field. User clicks on “Send” icon to send the message.	The system will display and error message “Message Empty!”	The system will display and error message “Message Empty!”

Specific Cases

Test ID	Current User	Friend's Ethereum account address	Message Content	Expected Result	Actual Result
1	User1	0x70997970C51812dc3A01 0C7d01b50e0d17dc79C8	"hi how are you?"	The message is sent.	The message is sent.

Test Results using Backend Scripts

The figure below shows the sending of messages between two registered User1 and User2. The message list of the user is also printed out to ensure the successful sending of message.



```
(base) melise@Melises-MBP BackEnd % yarn hardhat run scripts/send-message.js --network localhost
yarn run v1.22.19
$ '/Users/melise/Desktop/fyp/fyp/ChainTalk copy b/BackEnd/node_modules/.bin/hardhat' run scripts/send-message
.js --network localhost
----------------------------------------------------------------------------------------------------
Got message contract  at 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
message sent
My Messages: Result(1) [
  Result(6) [
    '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
    '0x70997970C51812dc3A010C7d01b50e0d17dc79C8',
    'hi how are you?',
    1711017241n,
    false,
    false
  ]
]
✨  Done in 2.56s.
```

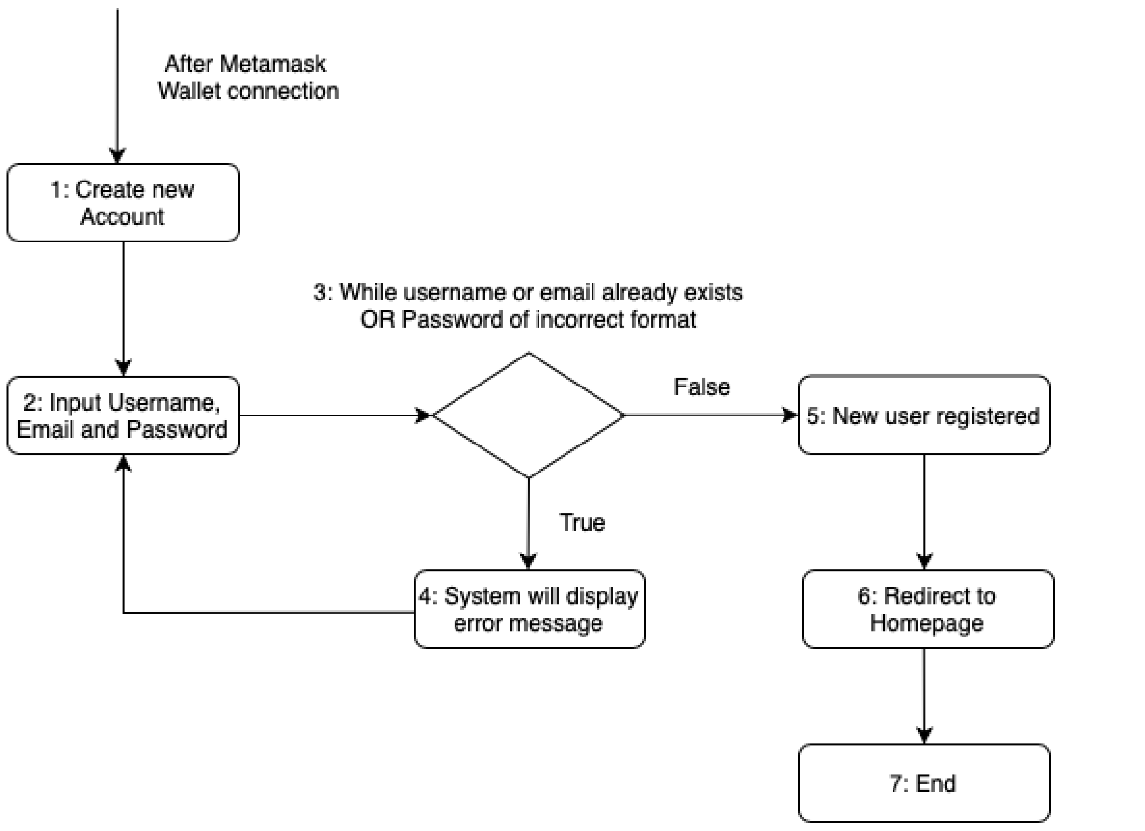
Figure 36: Sending of Message using Scripts

6.3 WHITE BOX TESTING

White box testing is a software testing method that examines the internal structure and workings of the system being tested. White box testing involves having detailed knowledge of the internal code, algorithms and implementation details of the software.

The below section displays the results after conducting white box testing on the different functionalities of ChainTalk.

6.3.1 USER REGISTRATION


After Metamask Wallet connection
1: Create new Account
2: Input Username, Email and Password
3: While username or email already exists OR Password of incorrect format
False
True
4: System will display error message
5: New user registered
6: Redirect to Homepage
7: End

Created using Drawio

Figure 37: User Registration White Box Testing

Basic Path	Expected Result	Status
1,2,3,5,6,7	User registered for an account successfully and is directed to the homepage after registration	Pass
1,2,3,4,2	User provided invalid input fields for the registration for an account. Error message is displayed.	Pass

6.3.2 ADD FRIEND

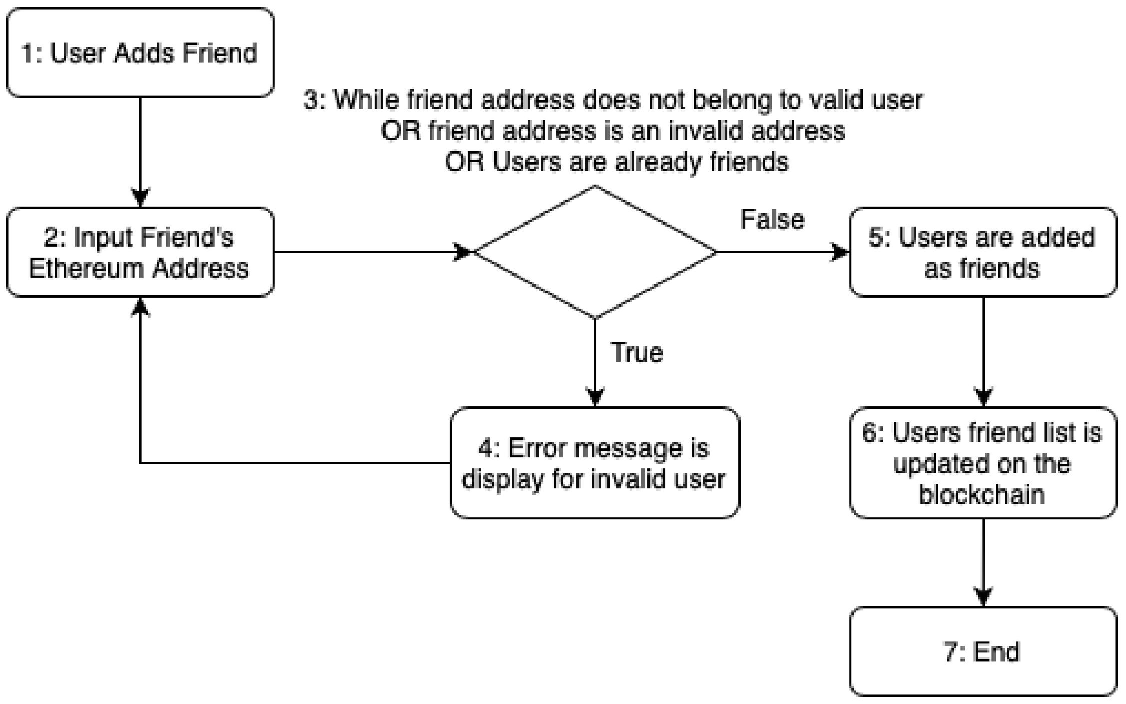
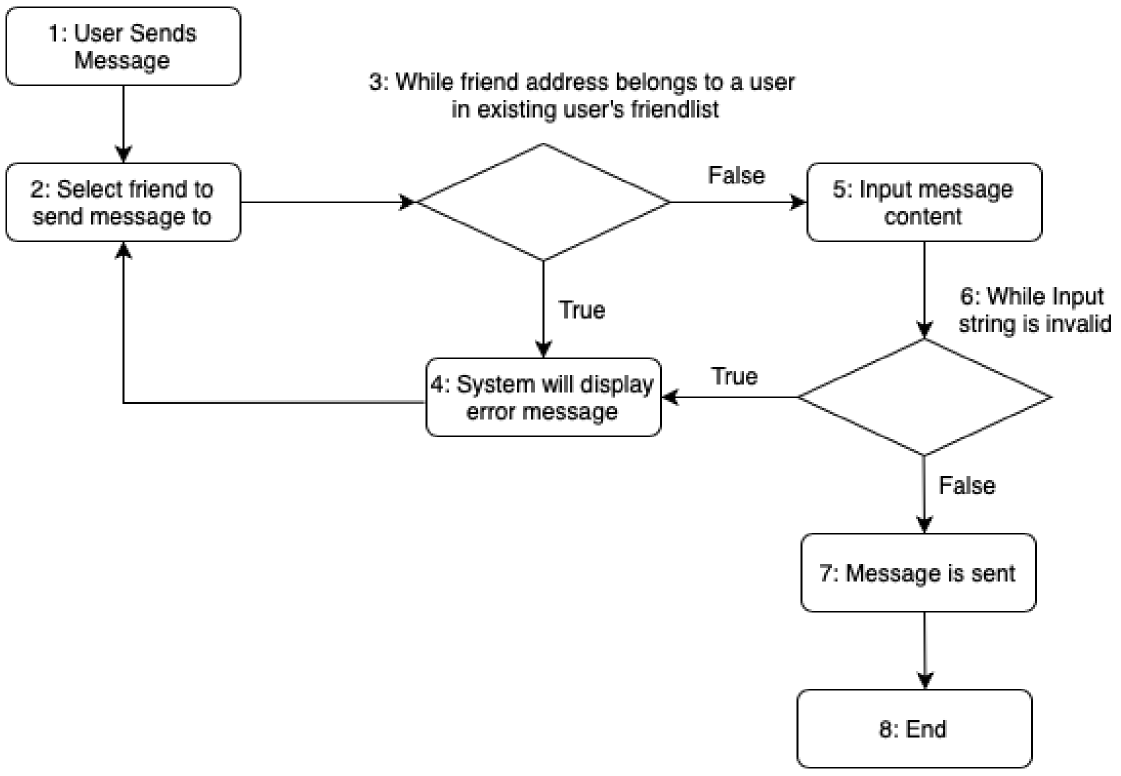

1: User Adds Friend
2: Input Friend's Ethereum Address
3: While friend address does not belong to valid user OR friend address is an invalid address OR Users are already friends
False
True
4: Error message is display for invalid user
5: Users are added as friends
6: Users friend list is updated on the blockchain
7: End

Figure 38: Add Friend White Box Testing *Created using Drawio*

Basic Path	Expected Result	Status
1,2,3,5,6,7	Users are successfully added as friends.	Pass
1,2,3,4,2	User provided invalid input fields for friend's address to add as friend. Error message is displayed.	Pass

6.3.3 SEND MESSAGE



1: User Sends Message
2: Select friend to send message to
3: While friend address belongs to a user in existing user's friendlist
False
5: Input message content
True
6: While Input string is invalid
4: System will display error message
True
False
7: Message is sent
8: End

Created using Drawio

Figure 39: Send Message White Box Testing

Basic Path	Expected Result	Status
1,2,3,5,6,7,8	Message is successfully being sent.	Pass
1,2,3,4,2	User provided invalid input fields for friend's address to send message to. Error message is displayed.	Pass
1,2,3,5,6,4,2	User provided invalid input string for message content to be sent. Error message is displayed.	Pass

Chapter 7 : FUTURE WORK & CONCLUSION

7.1 FUTURE WORK

Although using the decentralised IPFS file sharing system facilitates seamless, secure and low latency data distribution, it comes with its own share of limitations.

Serving as a database-as-a-service mode, there is a need to create a strong economic incentive However, as it is built as a community product that will benefit from everyone's contributions, no economic incentives were put in place. This results in IPFS becoming an impractical solution for long-term usage, especially for storing private and enterprise data [14]. Furthermore, as IPFS is inherently free, the nodes in the network are not volunteering to store a huge magnitude of private data on their systems for free for a long time. Hence, an economic incentive needs to be in place but is not the case for IPFS, resulting in limited storage capacity.

With the limitations of IPFS as stated above, there is a need for us to look into Layer 2 technologies to improve the efficiency and reduce the transaction time and cost for sending a message.

Integration of Layer 2 solutions such as state channels, side-chains, Optimistic Rolllups and ZK-Rollups can be considered as a potential future scalability enhancement for ChainTalk.

For a start, instead of the Sepolia testnet, the smart contracts can be deployed to the Polygon network which is a Layer 2 blockchain that is compatible with Ethereum.

7.2 CONCLUSION

In conclusion, this project has explored the landscape of decentralised messaging applications leveraging blockchain technology. The integration of the InterPlanetary File System (IPFS) as a decentralised storage solution has been identified as a pivotal aspect of the application. The content-addressed nature of IPFS ensures data integrity, while its peer-to-peer architecture aligns with the principle of blockchain technology.

Furthermore, the introduction of a digital identity and SBT enables user to have ownership over their own data, preventing identity theft. It also allows a user to be more credible and have its existence to be on the blockchain forever, removing the need for intensive identity verification.

In summary, this project lays the groundwork by addressing the identified research gaps, the proposed application aims to offer a secure, efficient and scalable messaging solution in a decentralised environment. This endeavour aligns with the broader trend towards leveraging blockchain for diverse applications and exemplifies the potential for decentralised technologies to reshape the landscape of secure communication.

Chapter 8 : REFERENCES

[1] Stacy Jo Dixon , “Statista,” Statista, 29 August 2023. [Online]. Available: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/.

[2] J. Anderson and R. Lee, “Pew Research Center,” 17 April 2018. [Online]. Available: https://www.pewresearch.org/internet/2018/04/17/concerns-about-the-future-of-peoples-well-being/.

[3] A. Chang, “Vox,” 2 May 2018. [Online]. Available: https://www.vox.com/policy-and-politics/2018/3/23/17151916/facebook-cambridge-analytica-trump-diagram.

[4] L. Rainie, “Americans’ complicated feelings about social media in an era of privacy concerns,” 27 March 2018. [Online]. Available: https://www.pewresearch.org/short-reads/2018/03/27/americans-complicated-feelings-about-social-media-in-an-era-of-privacy-concerns/.

[5] J. Lapienytė, “WhatsApp data leaked - 500 million user records for sale online,” 15 November 2023. [Online]. Available: https://cybernews.com/news/whatsapp-data-leak/.

[6] R. Blum and B. Betsy, “Achieving Digital Permanence: The many challenges to maintaining stored information and ways to overcome them,” *ACM Digital Library,* vol. 16, no. 6, p. 50, 2018.

[7] Statista Research Department, “Worldwide spending on blockchain solutions from 2017 to 2020, with forecasts for 2021 and 2024,” Statista, 27 October 2023. [Online]. Available: https://www.statista.com/statistics/800426/worldwide-blockchain-solutions-spending/.

[8] R. Yetis and K. S. Ozgur, “Blockchain Based Secure Communication for IoT Devices in Smart Cities,” *2019 7th International Istanbul Smart Grids and Cities Congress and Fair (ICSG),* 2019.

[9] S. Chaudhary, Rajesh Gupta, Riya Kakkar and Sudeep Tanwar, “A smart contract and IPFS-based framework for secure electric vehicles synchronization at charging station,” *Sustainable Energy, Grids and Networks,* vol. 37, no. 101272, 2024.

[10] E. G. W. V. B. Puja Ohlhaver, “Decentralized Society: Finding Web3's Soul,” May, 2022.

[11] Neo, “Medium: ERC-725 and ERC-735: What are they? New identity standards,” 22 January 2024. [Online]. Available: https://medium.com/leclevietnam/erc-725-and-erc-735-what-are-they-new-identity-standards-b50d751db039.

[12] J. Santos, “Hackernoon: First impressions with ERC 725 and ERC 735 — identity and claims,” 21 June 2018. [Online]. Available: https://hackernoon.com/first-impressions-with-erc-725-and-erc-735-identity-and-claims-4a87ff2509c9.

[13] N. Bozovic, “TechNadu,” 29 February 2020. [Online]. Available: https://www.technadu.com/keybase-review/93937/.

[14] “AWS: What is Ethereum?,” [Online]. Available: https://aws.amazon.com/blockchain/what-is-ethereum/.

[15] “IPFS Docs,” [Online]. Available: https://docs.ipfs.tech.

[16] “OpenZeppelin Docs,” [Online]. Available: https://docs.openzeppelin.com/contracts/5.x/.

[17] Arcana Network, “Medium,” Arcana Network Bloc, 3 January 2022. [Online]. Available: https://medium.com/arcana-network-blog/drawbacks-of-ipfs-a-brief-guide-to-understanding-ipfs-where-it-lacks-8be98a35e89e.

APPENDIX A

FULL CODE FOR CHAINTALK

The full code for the application can be found in the below GitHub Repository:

https://github.com/Melisepoon/ChainTalk.git