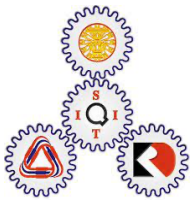


Blockchain Development - Group Assignment

Capstone Project

“Student fee CrowdFunding”

Submitted to

School of Information, Computer and Communication Technology
Sirindhorn International Institute of Technology
Thammasat University

By

Nutsuda Ploysopond	6222780049
Pimrumpha Chayapirad	6222770222
Thanyathon Saengsena	6222781492

Advisor

Dr. Watthanasak Jeamwatthanachai

Acknowledgement

We would like to convey our sincere gratitude to everyone who gave us the chance to finish this final project. We are also especially thankful to Dr. Watthanasak Jeamwatthanachai , one of our advisers, for his advice, suggestions, and helpful judgment of our effort. Without their assistance, this project will not succeed at this point. When we get into difficulty with this project, they offer an excellent solution. Last but not least, we appreciate the help from our family and group members.

Chapter1

Introduction

In today's digital age, blockchain technology has emerged as a game-changer, offering unprecedented transparency, security, and decentralization. By combining this cutting-edge technology with smart contracts, we have created a platform that ensures trust, accountability, and efficiency in the crowdfunding process.

Our platform serves as a bridge between students seeking financial assistance for their tuition fees and compassionate donors who want to make a positive impact on students' lives. Through our website, students can create compelling crowdfunding campaigns to showcase their academic aspirations and financial needs. Donors, on the other hand, have the opportunity to browse through a diverse range of student profiles, learn about their educational journeys, and contribute to the campaigns that resonate with them.

The power of blockchain technology plays a crucial role in building trust and transparency. Every donation and fund disbursement is recorded on the blockchain, forming an unchangeable and auditable record. Donors can see exactly how their contributions are used, promoting openness and responsibility.

Smart contracts, which rely on blockchain technology, automate the crowdfunding process without the need for middlemen. This means that funds are released to students like me when specific conditions are met. For example, if a campaign reaches its funding goal or when students achieve certain academic milestones, the funds can be disbursed.

By utilizing blockchain and smart contracts, our platform creates a secure and efficient environment that connects students in need with kind-hearted donors. At the same time, donors have the opportunity to directly impact our lives by helping us overcome financial obstacles and pursue our educational dreams.

1.1 Motivation

The current education system faces significant challenges in providing equal access to quality education due to financial barriers. Many deserving students are unable to pursue their educational goals due to limited financial resources. Traditional crowdfunding platforms have made some progress in addressing this issue, but they still suffer from certain limitations and inefficiencies. Therefore, there is a need to develop an education crowdfunding platform using blockchain technology to overcome these challenges and create a more inclusive and transparent system.

1.2 Objectives

The objective of implementing education crowdfunding is to enhance transparency, reduce transaction costs and delays, provide more accessibility, and enhance privacy and security concerns through the utilization of blockchain technology. By leveraging the inherent transparency of blockchain, we aim to create a platform where donors can have a clear view of how their funds are being utilized, promoting trust and accountability in the crowdfunding process. Additionally, by eliminating intermediaries and utilizing smart contracts, we can significantly reduce transaction costs and delays, ensuring that funds reach the intended recipients more efficiently. Our objective is to make education crowdfunding more accessible to a wider audience, breaking down geographical and financial barriers, and enabling individuals from all walks of life to contribute and support educational initiatives.

1.3 Problem statement

1.3.1 Lack of Transparency

Existing crowdfunding platforms often lack transparency in how funds are utilized and distributed. Donors are unable to track the progress and impact of their contributions, and students may not have a clear understanding of how funds are allocated. This lack of transparency erodes trust and makes it challenging to ensure accountability.

1.3.2 High Transaction Costs and Delays

Traditional crowdfunding platforms involve multiple intermediaries and payment processors, resulting in high transaction costs and delays in fund disbursement. These inefficiencies reduce the overall impact of donations and slow down the process of providing financial assistance to students in need.

1.3.3 Limited Accessibility

Many deserving students, particularly those from underserved regions, struggle to access crowdfunding opportunities due to geographical limitations and platform restrictions. Current platforms may focus on specific countries or regions, leaving out students who could benefit from global donor support. This limited accessibility hinders the potential impact of education crowdfunding efforts.

1.3.4 Privacy and Security Concerns

Centralized crowdfunding platforms often require users to disclose personal and financial information, which raises concerns about privacy and security. The storage and handling of sensitive data on these platforms may be vulnerable to hacking attempts and unauthorized access, compromising the privacy of donors and students.

1.4 Pain Points

1.4.1 Trust and Verification

All transactions and records are recorded on the blockchain, creating an immutable and transparent ledger. Donors can track and verify the flow of funds, ensuring that their contributions are used as intended. Additionally, student profiles and campaign details are stored securely on the blockchain, providing authenticity and transparency to potential donors.

1.4.2 Accessibility and User Experience

EduCrowd focuses on providing a user-friendly and accessible platform. Students and donors can easily register and login to their accounts, enabling them to participate in the crowdfunding process seamlessly. The platform is designed with intuitive navigation and clear instructions, ensuring a smooth user experience for both students and donors. Accessibility is prioritized, allowing users from different regions and backgrounds to access the platform and contribute to educational campaigns.

1.4.3 Scalability and Transaction Costs

By utilizing blockchain technology, EduCrowd reduces the need for intermediaries and streamlines the donation process, minimizing transaction costs. This cost-effectiveness benefits both students and donors, ensuring that a higher portion of the funds go directly towards supporting educational endeavors.

1.5 User and Benefits

Users of a student fee crowdfunding system, where students can raise funds for their education through a digital platform, can benefit in the following ways:

1.5.1 Students

Students who require financial support for their education can leverage the crowdfunding platform to raise funds from a wider audience. They can create personalized campaigns, share their stories, and receive contributions from individuals and organizations who are willing to support their educational journey.

1.5.2 Donors

Individuals or organizations interested in supporting students' education can easily browse through various campaigns on the crowdfunding platform and make direct contributions to the students they resonate with. Donors can choose specific students or campaigns to support and have the satisfaction of making a positive impact on someone's education.

Chapter2

Requirements Specification

2.1 System Description

The student fee crowdfunding system is a digital platform that connects students in need of financial support for their education with individuals and organizations willing to contribute funds. The system aims to provide an accessible and transparent way for students to raise the necessary funds to pursue their educational goals.

The system allows students to create personalized campaigns on the platform, sharing their stories, educational aspirations, and financial needs. They can provide details about their academic pursuits, tuition fees, living expenses, and any other relevant information to attract potential donors.

Donors can browse through the various student campaigns on the platform and choose the ones they wish to support. They can make direct contributions to specific students or allocate funds to a general pool where students can request financial assistance. The system provides multiple payment options to accommodate donors' preferences, such as credit/debit cards, bank transfers, or cryptocurrency payments.

To ensure transparency and accountability, the system leverages blockchain technology. Each donation transaction is recorded on the blockchain, creating an immutable and transparent ledger of all financial activities. This allows donors to track how their contributions are utilized and provides assurance that funds are allocated appropriately.

2.2 Requirements

2.2.1 User Registration and Authentication

User Registration involves the process where users provide their personal details, such as name, email, and password, which are recorded on the blockchain. Verifies the user's identity when accessing the web application.

2.2.2 Payment Integration

Integrate payment gateways to enable secure and convenient transactions between donors and students. Users should be able to make contributions using various payment methods, such as credit cards, digital wallets, or cryptocurrencies.

2.2.3 Student Profile Management

Provide a feature for students to create and manage their profiles. This includes personal details, academic information, and the amount of tuition fees they need to cover. Students should have the ability to edit their profiles, track their progress, and communicate with donors.

2.2.4 Crowdfunding Campaign Management

Enable students to launch crowdfunding campaigns to gather support for their tuition fees. Students can set financial goals, provide details about their education, and share their stories to attract donors. The platform should display the progress of each campaign, including the total amount raised and the remaining target.

2.3 Main Features

2.3.1 Campaign Registration

Our platform allows students to easily create compelling crowdfunding campaigns to showcase their educational aspirations and financial needs. Students can provide details about their academic pursuits, career goals, and the amount of funding required to support their education.

2.3.2 Donation system

The platform facilitates direct engagement between students and donors, creating a connection that goes beyond financial contributions. Donors can browse through student profiles, learn about their educational journeys, and interact with them through messaging or commenting features.

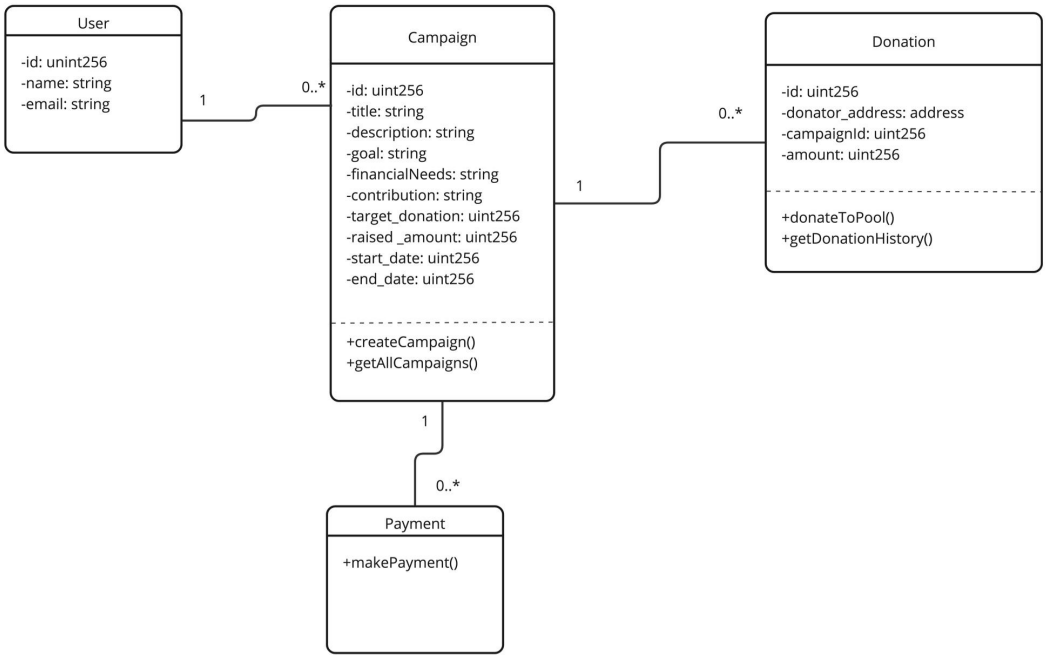
2.3.3 Transparent Fund Tracking

Transparency is a key aspect of our platform. Donors can track the progress of each campaign, including the total funds raised, remaining target, and how the funds are utilized. This fosters trust and accountability, giving donors confidence in their contributions.

2.3.4 Secure Payment Processing

Our platform incorporates secure payment processing mechanisms, allowing donors to make contributions using various methods using digital wallets, or cryptocurrencies.

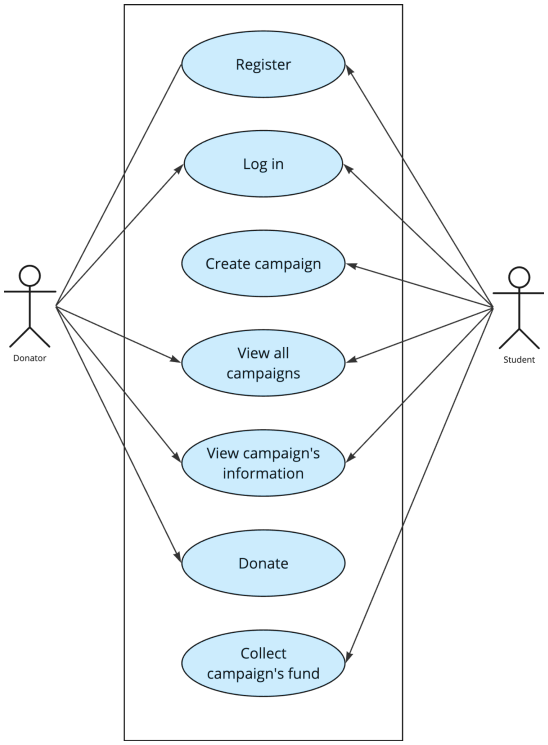
2.4 System Design



Chapter3

Design Specification

3.1 Use Case Diagram



In our system, we have provided functionality for both users and donors to interact with the platform. Users and donors can perform various actions such as registration, login, viewing all campaigns, and accessing campaign information. Additionally, donors have the ability to donate to a campaign, while students can collect funds for their campaigns.

The registration feature allows users and donors to create an account on our platform. By providing necessary information, such as their name, email, and password, they can sign up and gain access to the system. This ensures that each user and donor has a unique identity within our system.

Once registered, users and donors can log in using their credentials. This authentication process ensures that only authorized individuals can access their respective accounts and perform actions on the platform.

To enable users and donors to explore the campaigns available, we have implemented a feature that allows them to view all the campaigns. This provides an overview of the various campaigns available on the platform, including information such as the campaign name, description, and current status.

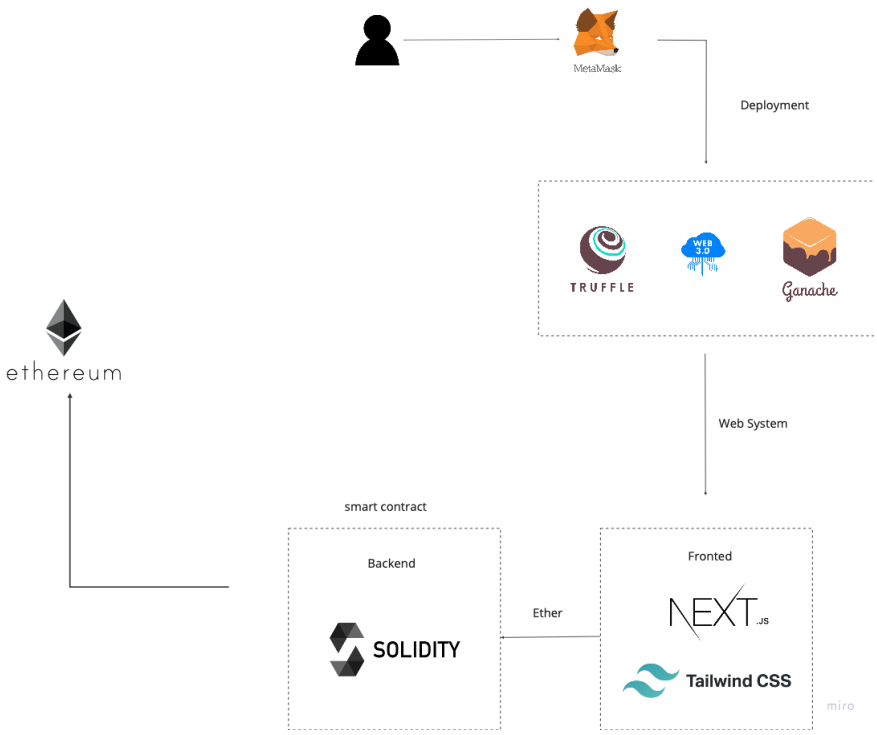
Furthermore, users and donors can access detailed information about each campaign. This includes additional details such as the campaign goal, progress, and any updates or milestones achieved. By providing comprehensive campaign information, we aim to enable users and donors to make informed decisions regarding their contributions.

One of the key functionalities of our system is the ability for donors to contribute funds to campaigns. Donors can select a campaign they wish to support and make a donation, typically in the form of monetary contributions. This feature allows donors to actively participate in the campaigns they find meaningful and contribute to their success.

On the other hand, students who create campaigns can collect funds from the donors. By initiating a campaign and setting a specific goal, students can present their projects or initiatives to potential donors. As donors contribute funds to the campaign, students can track the progress towards their goal and collect the funds raised to support their projects.

3.2 Tech Stacks

The technologies we will use.



Overview System

The system utilizes ReactJS as the front-end and NodeJS as the back-end. Solidity is used for contract development, and the contract is compiled into JSON-formatted ABI code using the solc npm package. The ABI interface is then used to deploy the contract using a Web3 provider instance. Instead of using a local node, Infura is used as a remote node to connect to the Ethereum network.

To get started with the system, users need to set up the Metamask cryptocurrency wallet, which is a browser extension that enables interaction with decentralized applications (dApps). Once a user creates an account in Metamask and transfers Ethereum to their account, they can interact with the system. Metamask injects a Web3 instance into the web browser, facilitating the interaction.

Users can create campaigns, and other users can contribute to those campaigns. Campaign managers can also create requests to specify how the collected funds will be used. Contributors can review and approve these expense requests, and if approved by the majority of backers, the Ether is sent to the vendors. [1]

Metamask is a browser extension that acts as a cryptocurrency wallet and a bridge between web browsers and the Ethereum blockchain. In the system architecture, Metamask serves as a crucial component for interacting with decentralized applications (dApps) and handling Ethereum transactions. It provides users with the ability to create and manage Ethereum accounts, securely store private keys, and sign transactions.

Deployment technology

1. Truffle: Truffle is a development framework for Ethereum that provides a suite of tools to compile, deploy, and test smart contracts. It simplifies the process of building and deploying decentralized applications.
2. Ganache: Ganache is a local Ethereum blockchain for development and testing purposes. It provides a set of accounts with test Ether, mimicking the behavior of a live Ethereum network.
3. Web3.js: Web3.js is a JavaScript library that provides an interface for interacting with the Ethereum blockchain. Once your contracts are deployed, you can use Web3.js to interact with and call the functions of your deployed contracts from your client-side applications. Web3.js allows you to send transactions, read contract states, and listen for events emitted by the contracts.

In the development of our system, we have divided it into two main parts: the frontend and the backend.

For the frontend, we have chosen to use Next.js and Tailwind CSS.

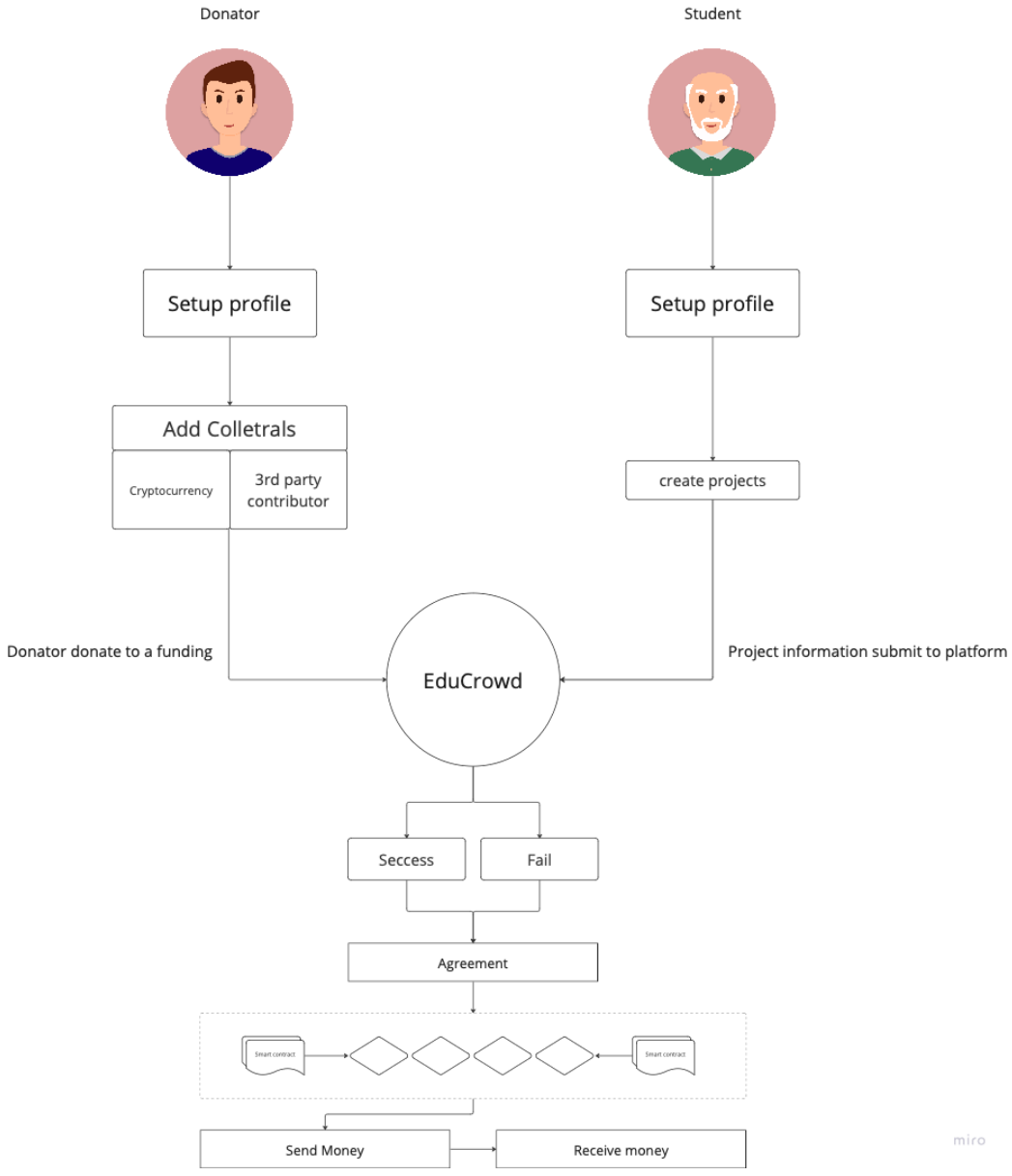
1. Next.js is a popular React framework that allows us to build powerful and scalable frontend applications. It provides server-side rendering, automatic code splitting, and easy routing, among other features. With Next.js, we can create fast and optimized web pages that deliver a smooth user experience.
2. Tailwind CSS is a utility-first CSS framework that helps us rapidly build custom user interfaces. It promotes a responsive and mobile-first approach, making it easier to create visually appealing and responsive designs.

For the backend of our system, we have decided to utilize Solidity as the programming language to write our smart contract.

Once our smart contract is written in Solidity, we can compile it using tools like the Solidity compiler (solc) or the Truffle framework. The compiled contract will produce a bytecode representation that can be deployed to the Ethereum network.

Solidity's integration with Ethereum allows our smart contract to interact with other contracts, send and receive Ether, and emit events that can be observed by external systems. This opens up a wide range of possibilities for building decentralized applications and implementing blockchain-based solutions.

3.3 Software design



Chapter4

Implementation

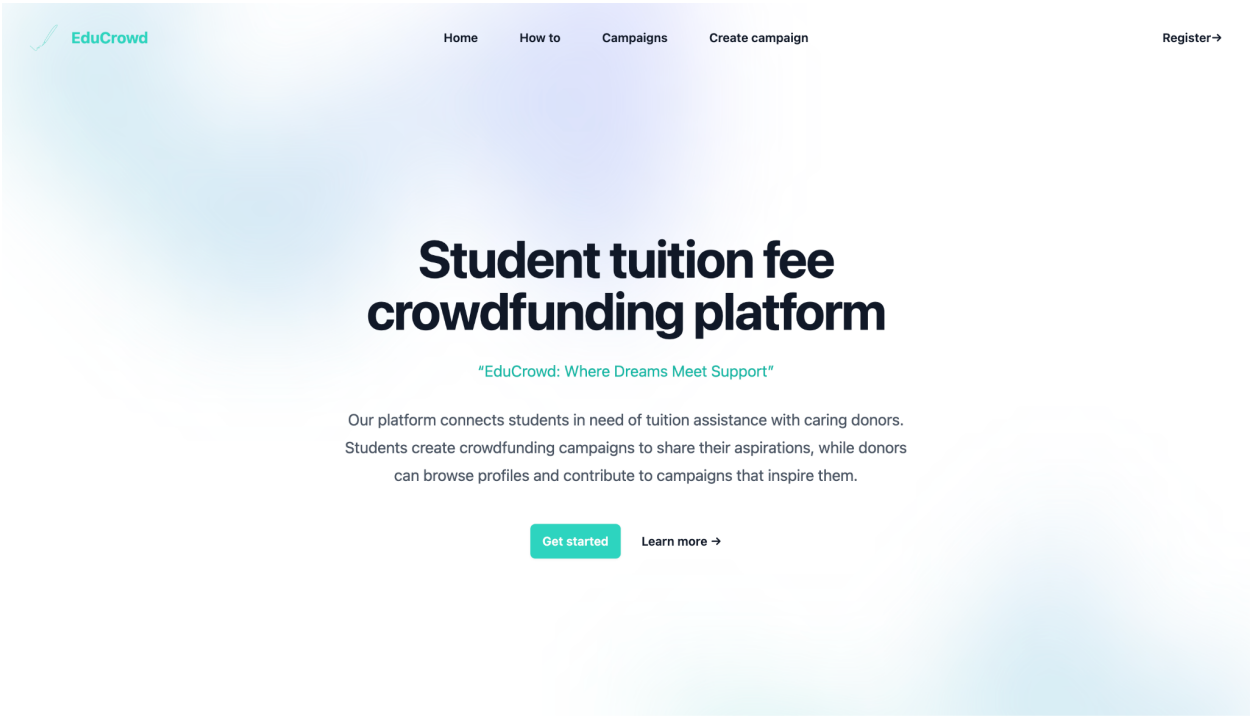
4.1 User Experience

In our system, we have focused on creating a user-friendly interface to ensure a seamless user experience.

4.1.1 Homepage

The homepage serves as the entry point where users can learn about our system and its features. It provides a brief overview of the platform and its purpose, helping users understand its functionality.

One of the key of this pages we have designed is the "View All Campaigns" page. Here, anyone can access and browse through all the campaigns available on our platform. Each campaign is displayed with relevant information, such as the campaign name, description. This allows users to explore and discover campaigns that align with their interests.



What to expect

1

Start with the basics

Kick things off with your name and location.

2

Tell your story

We'll guide you with tips along the way.

3

Share with friends and family

People out there want to help you.

Current Campaigns

Find the campaign that interests you

0.0/0.4

May 31, 2023

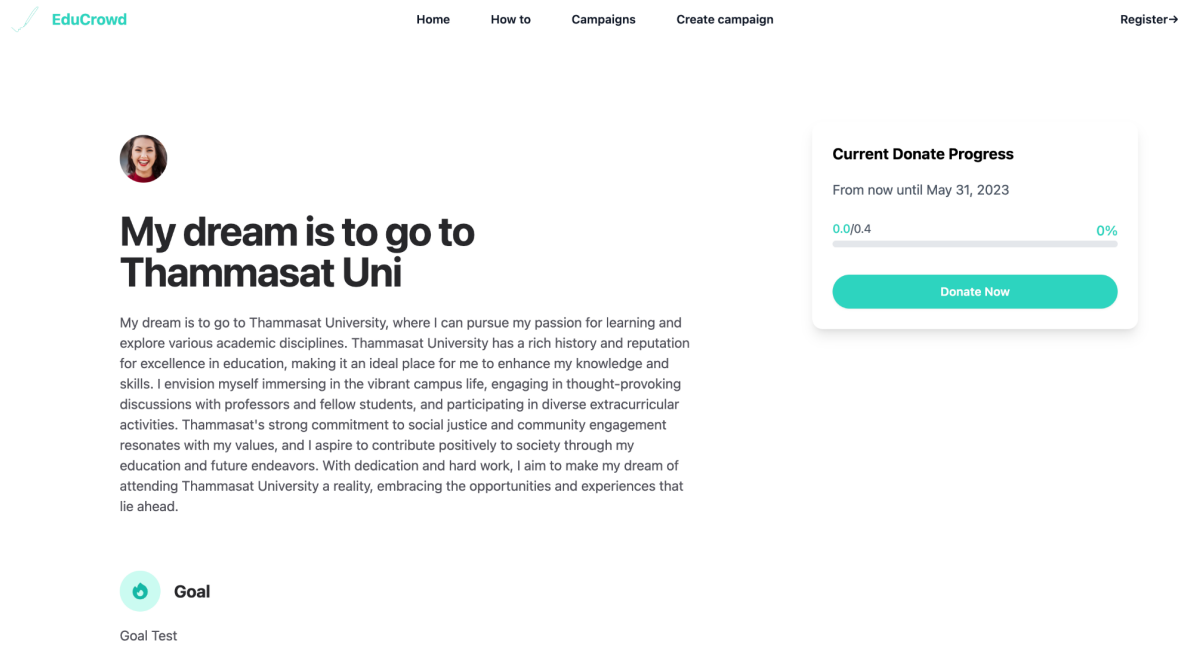
My dream is to go to Thammasat Uni

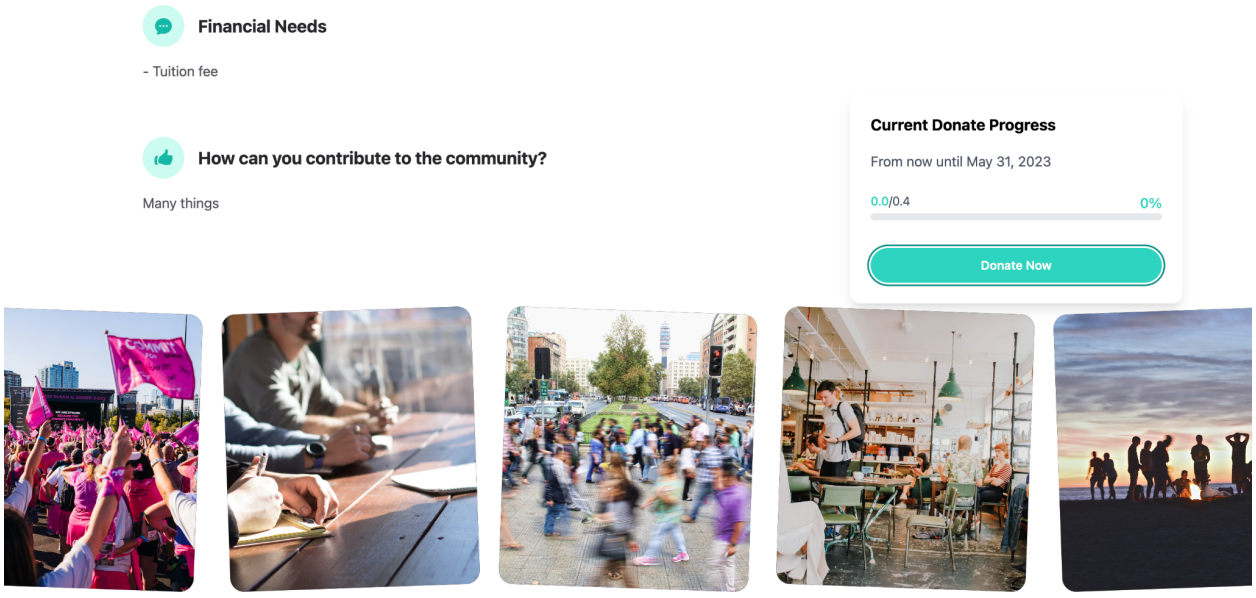
My dream is to go to Thammasat University, where I can pursue my passion for learning and explore various academic disciplines. Thammasat University has a rich...

4.1.2 Campaign Information page

To provide detailed information about each campaign, we have designed a dedicated "Campaign Information" page. This page displays comprehensive details about a specific campaign, including its objective, target amount, deadline, and any updates or milestones achieved. By presenting this information in a clear and organized manner.

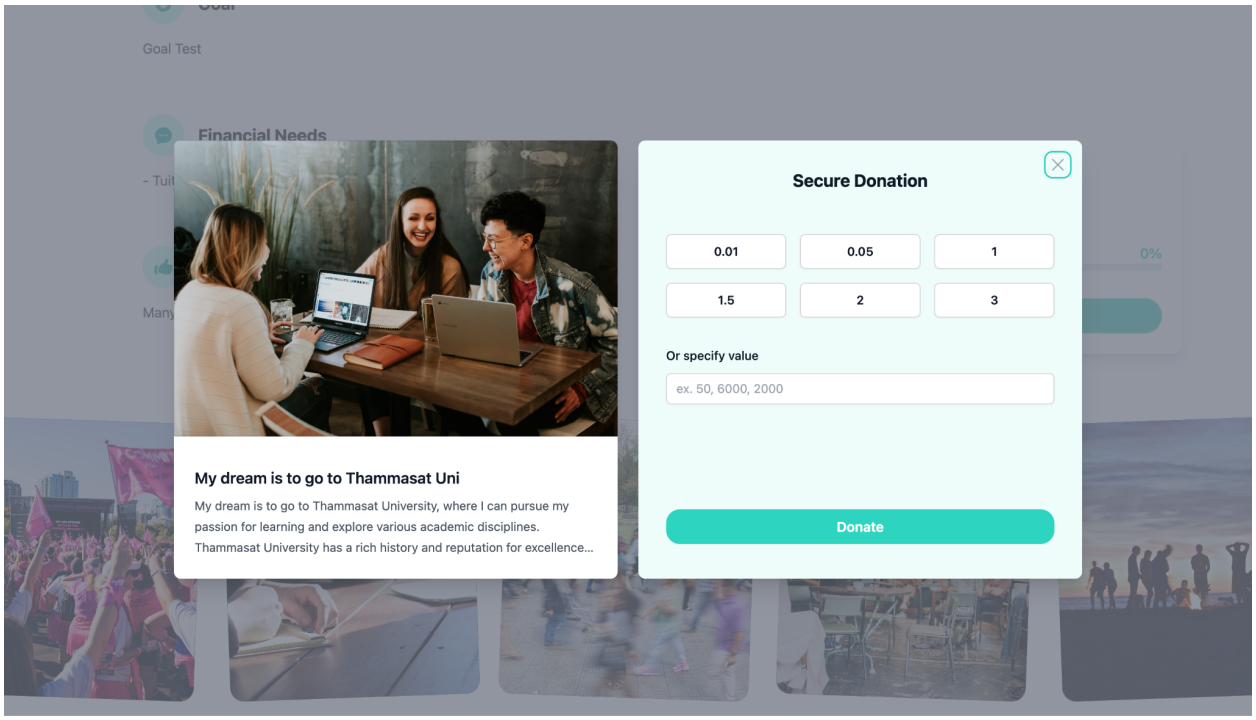
To showcase the progress of each campaign, we have included a "Current Donation Progress" section. This section visually represents the amount of donations received compared to the campaign's target. Users can easily track the campaign's progress and see how close it is to reaching its goal. This feature creates transparency and encourages user engagement.





4.1.3 Payment page

In the payment page, we have designed a user-friendly interface for making donations. Users can choose to donate by clicking on pre-defined donation amounts, which the system provides based on common values. Additionally, we have included an option for users to specify their own donation amount, giving them the flexibility to contribute according to their personal preferences. This feature empowers users to make a donation that fits their financial capabilities.



4.1.4 Create campaign page

For student users, we have designed a "Create Campaign" page that allows them to initiate their own campaigns. This page includes form fields where students can provide essential details about their project or initiative. They can specify the campaign's title, description, fundraising goal, and deadline. By designing this page, we aim to encourage student involvement and support their fundraising efforts.

EduCrowd

Home How to Campaigns Create campaign Register→

Create a new campaign

This information will be displayed publicly so be careful what you share.

What is the campaign title?

How much do you need?

ETH 0

Give a description to your campaign.

Begin with a compelling introduction that grabs the reader's attention and highlights your motivation for applying to the institution.

Explain about your financial needs and what do you need the fund for

What is your goal?

Clearly state your educational and career goals, explaining how it can impact the society.

How can you contribute to the community?

Tell donators about how you can contribute to the community. It can be anything.

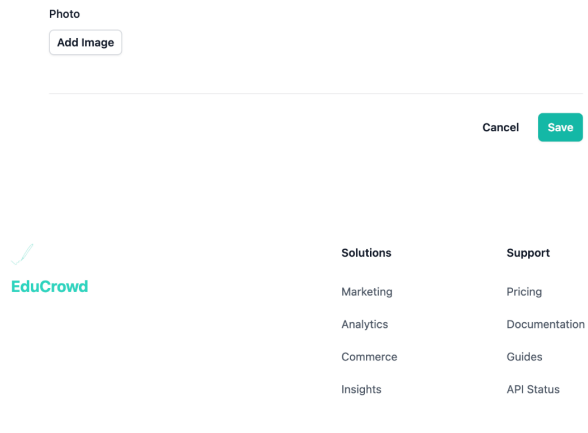
Campaign date

Start date End date

Tell donators about how you can contribute to the community. It can be anything.

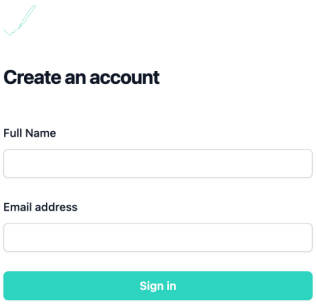
Photo

Add Image



4.1.5 Create and Sign in account page

Our sign-in and sign-up account design prioritizes simplicity, ease of use, and security. The user interface for creating an account is intuitive, guiding users through a straightforward process. To sign up, users provide their basic information, including name, email address, and a secure password. We ensure data accuracy by implementing validation checks on the provided information.



4.2 Build the System

The system that we will build are

1. Payment Integration: Connecting Ethereum wallets to a website
2. User Registration and Authentication
3. Crowdfunding Campaign Management: Storing campaign
4. Student Profile Management

We will utilize the following technologies.

1. JavaScript / React / Next.js
2. Ganache / Truffle (to establish a local blockchain for testing the app)
3. MetaMask (to create a personal Ethereum wallet)

4.2.1 Ethereum Development Setup

1. Metamask
2. Ganache
3. Truffle

4.2.2 NextJS Development Setup

4.2.3 Writing Smart Contracts

1. **Campaign Registration Contract**

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./UserRegistration.sol";

contract CampaignRegistration is UserRegistration {
    struct Campaign {
        uint256 id;
        string title;
        string description;
        string goal;
        string financial_needs;
        string contribution;
        uint256 target_donation;
        uint256 raised_amount;
        uint256 start_date;
        uint256 end_date;
        bool isActive;
        User owner;
    }

    mapping(uint256 => Campaign) public campaigns;
    uint256 public campaignIdCounter;

    event CampaignCreated(
        uint256 indexed campaignId,
        string title,
        address indexed beneficiary
    );

    event CampaignClosed(
        uint256 indexed campaignId,
        string title,
        address indexed beneficiary
    );
```

```
37
38      function createCampaign(
39          string memory _title,
40          string memory _description,
41          string memory _goal,
42          string memory _financial_needs,
43          string memory _contribution,
44          uint256 _targetDonation,
45          uint256 _start_date,
46          uint256 _end_date
47      ) public {
48          require(isRegistered(), "Unauthorized");
49          require(bytes(_title).length > 0, "Title field is required");
50          require(
51              bytes(_description).length > 0,
52              "Description field is required"
53          );
54          require(bytes(_goal).length > 0, "Goal field is required");
55          require(
56              bytes(_financial_needs).length > 0,
57              "Financial needs infomation is required"
58          );
59          require(
60              bytes(_contribution).length > 0,
61              "Contribution field is required"
62          );
63          require(_targetDonation > 0, "Target donation is required");
64          require(
65              _start_date >= block.timestamp,
66              "Start date is less than end date"
67          );
68          require(_end_date > _start_date, "End date is less than start date");
69
70          campaignIdCounter++;
71
72          campaigns[campaignIdCounter] = Campaign({
73              id: campaignIdCounter,
74              title: _title,
75              description: _description,
76              goal: _goal,
77              financial_needs: _financial_needs,
78              contribution: _contribution,
79              raised_amount: 0,
80              target_donation: _targetDonation,
81              start_date: _start_date,
82              end_date: _end_date,
83              isActive: true,
84              owner: users[msg.sender]
85          });
86          emit CampaignCreated(campaignIdCounter, _title, msg.sender);
87      }
```

```
88
89      function getAllCampaigns() public view returns (Campaign[] memory) {
90          uint256 totalCampaigns = campaignIdCounter;
91          Campaign[] memory allCampaigns = new Campaign[](totalCampaigns);
92
93          uint256 index = 0;
94          for (uint256 i = 1; i <= totalCampaigns; i++) {
95              Campaign storage campaign = campaigns[i];
96              if (campaign.isActive) {
97                  allCampaigns[index] = campaign;
98                  index++;
99              }
100         }
101         return allCampaigns;
102     }
103 }
```

This is a Solidity smart contract code for a Campaign Registration contract that inherits from a User Registration contract. Let's go through the code step by step:

1. The contract is defined with the name "CampaignRegistration" and it imports another contract called "UserRegistration". The contract uses the Solidity version ^0.8.0 and specifies the SPDX-License-Identifier as MIT.

2. Inside the contract, a struct named "Campaign" is defined to store information about each campaign. It includes various properties such as the campaign ID, title, description, financial goals, contribution information, target donation amount, raised amount, start and end dates, activity status, and owner address.

3. A mapping named "campaigns" is declared to map campaign IDs to their corresponding Campaign struct.

4. A variable "campaignIdCounter" is defined to keep track of the total number of campaigns created.

5. Two events, "CampaignCreated" and "CampaignClosed", are defined to emit events when a campaign is created or closed.

6. The function "helloWorld()" is a simple view function that returns the string "hello world". It doesn't modify the state of the contract and can be called by anyone.

7. The function "createCampaign()" is used to create a new campaign. It takes various parameters such as the campaign title, description, goals, financial needs, contribution information, target donation amount, and start/end dates. It performs various require statements to validate the input parameters. If the validation passes, a new campaign is created and stored in the "campaigns" mapping. An event "CampaignCreated" is emitted.

8. The function "getAllCampaigns()" is a view function that returns an array of all active campaigns. It iterates through the campaigns stored in the mapping and filters out inactive campaigns. The resulting array is returned.

Overall, this contract allows users to create campaigns, stores campaign information, and provides a way to retrieve all active campaigns. It inherits functionality from the "UserRegistration" contract and can be used as a foundation for a decentralized crowdfunding application.

2. Donation System Contract

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./CampaignRegistration.sol";
import "./PaymentSystem.sol";

contract DonationSystem is CampaignRegistration, PaymentSystem {
    struct Donation {
        uint256 id;
        address donator_address;
        uint256 campaignId;
        uint256 amount;
        bool is_refunded;
        bool is_transfered_to_owner;
    }

    mapping(uint256 => Donation) public donations;
    uint256 public donateCount;
    address private constant DONATION_POOL =
        0x3b46755B8DB3EE2758620f084b9dBAd65b5Ac73b;
    event DonationMade(
        uint256 indexed campaignId,
        address indexed userId,
        uint256 amount
    );

    function donateToPool(
        uint256 _campaign_id,
        uint256 _amount
    ) public payable {
        require(_amount > 0, "Amount must be greater than zero");
        require(isRegistered(), "User not registered");
        require(
            msg.sender.balance > _amount,
            "Amount must be greater than zero"
        );
        require(campaigns[_campaign_id].id > 0, "Campaign does not exist");

        donateCount++;

        donations[donateCount] = Donation({
            id: donateCount,
            donator_address: msg.sender,
            campaignId: _campaign_id,
            amount: _amount,
            is_refunded: false,
            is_transfered_to_owner: false
        });

        campaigns[_campaign_id].raised_amount += _amount;
        makePayment(_amount, DONATION_POOL);
        emit DonationMade(_campaign_id, msg.sender, _amount);
    }
```

```
55
56      function closeCampaign(uint256 _campaignId) public {
57          require(_campaignId <= campaignIdCounter, "Invalid campaign ID");
58          Campaign storage campaign = campaigns[_campaignId];
59          require(campaign.isActive, "Campaign is not active");
60
61          Donation[] memory history = getDonationHistory(_campaignId);
62          if (campaign.raised_amount >= campaign.target_donation) {
63              //Success campaign
64              makePayment(campaign.raised_amount, campaign.owner.user_address);
65              for (uint256 i = 1; i <= history.length; i++) {
66                  history[i].is_transfered_to_owner = true;
67              }
68          } else {
69              require(
70                  block.timestamp >= campaign.end_date,
71                  "Campaign end date not reached"
72              );
73              // Fail campaign => Refund the donated amount to the donators
74              for (uint256 i = 1; i <= history.length; i++) {
75                  makePayment(history[i].amount, history[i].donator_address);
76                  history[i].is_refunded = true;
77              }
78          }
79          campaign.isActive = false;
80          emit CampaignClosed(campaign.id, campaign.title, msg.sender);
81      }
82
83      function getDonationHistory(
84          uint256 _campaignId
85      ) public view returns (Donation[] memory) {
86          require(_campaignId <= campaignIdCounter, "Invalid campaign ID");
87          Donation[] memory donationHistory = new Donation[](donateCount);
88          for (uint256 i = 1; i <= donateCount; i++) {
89              if (donations[i].id == _campaignId) {
90                  donationHistory[i] = donations[i];
91              }
92          }
93          return donationHistory;
94      }
95  }
```

This code represents a smart contract called "DonationSystem" that is built on top of the "CampaignRegistration" and "PaymentSystem" contracts. It introduces functionality related to making donations to campaigns and managing the donation history.

The contract includes the following key elements:

1. Structs: It defines a struct called "Donation" that represents a single donation made by a user. It includes properties such as the donation ID, donator address, campaign ID, donation amount, and flags indicating whether the donation has been refunded or transferred to the campaign owner.

2. Mapping: The contract maintains a mapping called "donations" that maps donation IDs to Donation objects. It allows for easy retrieval of donation information based on the donation ID.

3. Event: The contract emits a "DonationMade" event whenever a donation is made to a campaign. The event includes the campaign ID, user ID, and the amount of the donation.

4. Donation Functions: The contract provides a function called "donateToPool" that allows users to make a donation to a specific campaign. The function requires the user to specify the campaign ID and the amount to donate. It checks various conditions such as the amount being greater than zero, user registration, and campaign existence. It then increments the donation count, creates a new Donation object, updates the raised amount of the campaign, and makes a payment to the donation pool using the "makePayment" function inherited from the PaymentSystem contract. Finally, it emits the "DonationMade" event.

5. Campaign Closing: The contract includes a function called "closeCampaign" that allows the campaign owner to close a campaign. It requires the campaign ID as input. The function checks if the campaign is active and if the end date has been reached. If the campaign has reached its target donation amount, it makes a payment to the campaign owner and marks the donations as transferred. If the campaign has failed or the end date has been reached, it refunds the donated amount to the donors and marks the donations as refunded. The function also sets the campaign status to inactive and emits the "CampaignClosed" event.

6. Donation History: The contract provides a function called "getDonationHistory" that retrieves the donation history for a specific campaign. It takes the campaign ID as input and returns an array of Donation objects representing the donation history for that campaign.

Overall, this contract extends the functionality of the CampaignRegistration and PaymentSystem contracts by introducing donation-related operations, tracking donation history, and handling campaign closures based on donation success or failure.

3. Payment System Contract

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract PaymentSystem {
    function makePayment(uint256 _amount, address _recipant) public payable {
        require(msg.value == _amount, "Invalid Payment");
        (bool success, ) = _recipant.call{value: _amount}("");
        require(success, "Payment failed");
    }
}
```

This code represents a smart contract called "PaymentSystem" that provides a function for making payments.

The contract includes the following key elements:

1. Function: The contract defines a function called "makePayment" that allows for making payments to a recipient. It takes two parameters: the payment amount (_amount) and the recipient's address (_recipient).

2. Payment Validation: The function checks if the value sent with the transaction (msg.value) is equal to the specified payment amount (_amount). If they are not equal, it throws an exception with the error message "Invalid Payment". This ensures that the correct payment amount is provided.

3. Payment Execution: If the payment amount is valid, the function uses the recipient's address (_recipient) to call the fallback function of the recipient contract and transfer the payment amount. The "call" function is used to execute the contract call and returns a tuple (success, data) indicating the success status of the call.

4. Success Validation: The function checks if the payment execution was successful by validating the "success" boolean variable returned from the call. If the payment execution failed, it throws an exception with the error message "Payment failed".

Overall, this contract provides a simple payment functionality that ensures the correct payment amount is provided and attempts to transfer the payment amount to the specified recipient address.

4. User Registration Contract

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract UserRegistration {
    struct User {
        uint256 id;
        string name;
        string email;
        address user_address;
    }

    mapping(address => User) public users;

    event UserRegistered(address indexed userAddress, uint256 indexed userId);

    uint256 public userIdCounter;

    function registerUser(string memory _name, string memory _email) public {
        require(msg.sender != address(0), "Invalid user address");
        require(users[msg.sender].id == 0, "User already registered");
        require(bytes(_name).length > 0, "Name is required");
        require(bytes(_email).length > 0, "Email is required");
        userIdCounter++;
        users[msg.sender] = User({
            id: userIdCounter,
            name: _name,
            email: _email,
            user_address: msg.sender
        });
        emit UserRegistered(msg.sender, userIdCounter);
    }

    function getUserAddress() public view returns (address) {
        return msg.sender;
    }

    function isRegistered() public view returns (bool) {
        return users[msg.sender].id != 0;
    }
}
```

This code represents a smart contract called "UserRegistration" that provides user registration functionality for a decentralized application.

The contract includes the following key elements:

1. Struct: The contract defines a struct called "User" that represents the user information. It includes three properties: id (unique identifier for the user), name, and email.

2. Mapping: The contract uses a mapping called "users" to store user information based on their Ethereum address. Each user's Ethereum address is associated with their corresponding User struct.

3. Event: The contract defines an event called "UserRegistered" that is emitted when a user is successfully registered. It includes the user's Ethereum address and their assigned user ID.

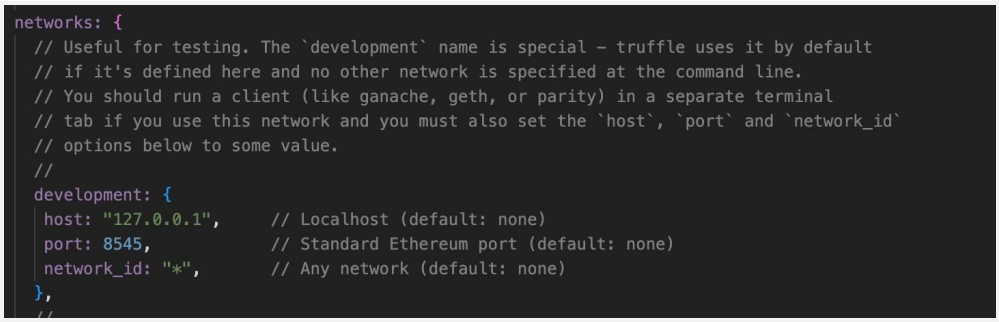
4. Counter: The contract maintains a userIdCounter variable to assign unique IDs to each registered user. It starts from 1 and increments for each new user.

5. User Registration Function: The contract provides a function called "registerUser" that allows a user to register by providing their name and email. The function checks if the user is already registered by verifying if their ID is present in the users mapping. It also requires non-empty values for the name and email fields. If the user is not already registered and the provided information is valid, a new User struct is created, associated with the user's Ethereum address, and the UserRegistered event is emitted.

6. User Registration Check Function: The contract provides a function called "isRegistered" that allows other functions to check if the calling user is already registered. It returns a boolean value indicating the registration status.

Overall, this contract enables users to register by providing their name and email, and it maintains a mapping of registered users for easy lookup. The UserRegistration contract can be imported and used by other contracts that require user registration functionality.

4.3 Deployment and Maintenance



```
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache, geth, or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //
  development: {
    host: "127.0.0.1",     // Localhost (default: none)
    port: 8545,            // Standard Ethereum port (default: none)
    network_id: "*",       // Any network (default: none)
  },
```

set the port to be the same as Ganache (assuming Ganache is running on port 8545)

Go to the root directory and run in the terminal with this code .

```
truffle migrate
```

Then on local blockchain will deploy

By following these steps, your smart contract will be deployed to your local blockchain. This deployment process will ensure that your contract is operational and ready to use. Upon successful deployment, you will receive the output message confirming the completion of the deployment process as follows.

```
   Replacing 'CampaignRegistration'
   --------------------------------
   > transaction hash:    0x529bbe7d09c65a43ddc87ec6a4d9c1e16213e9ca0d74f9b11eaa7b2e6a63181c
   > Blocks: 0            Seconds: 0
   > contract address:    0xb0d39BeBee57d07acF77991fA5C324122E1068e6
   > block number:        68
   > block timestamp:     1684739512
   > account:             0x3b46755B8DB3EE2758620f084b9dBAd65b5Ac73b
   > balance:             76.891078158511000278
   > gas used:            1986383 (0x1e4f4f)
   > gas price:           2.500583808 gwei
   > value sent:          0 ETH
   > total cost:          0.004967117166286464 ETH


   Replacing 'DonationSystem'
   --------------------------
   > transaction hash:    0x2c04470f0d75a01096006bda6c192d900c2b7d99696f64744957c79261ef8501
   > Blocks: 0            Seconds: 0
   > contract address:    0x8800a6b3cfbfE4d47f1916FEEd659529B05c697C
   > block number:        69
   > block timestamp:     1684739512
   > account:             0x3b46755B8DB3EE2758620f084b9dBAd65b5Ac73b
   > balance:             76.883678454212246866
   > gas used:            2959226 (0x2d277a)
   > gas price:           2.500553962 gwei
   > value sent:          0 ETH
   > total cost:          0.007399704298753412 ETH


   Replacing 'PaymentSystem'
   -------------------------
   > transaction hash:    0x0a31e6029e5f69cc917d73939e524136c220b125e1e44f36a0844ee65cbdb72e
   > Blocks: 0            Seconds: 0
   > contract address:    0x03e6Bd97c7a46F24EE7E13C34ffbf44054BE2520
   > block number:        70
   > block timestamp:     1684739512
   > account:             0x3b46755B8DB3EE2758620f084b9dBAd65b5Ac73b
   > balance:             76.883072962077898226
   > gas used:            242144 (0x3b1e0)
   > gas price:           2.500545685 gwei
   > value sent:          0 ETH
   > total cost:          0.00060549213434864 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:     0.014862524735517787 ETH

Summary
=======
> Total deployments:   4
> Final cost:          0.014862524735517787 ETH
```

```
> Artifacts written to /Users/Earn/Desktop/blockchain/web/dapp/build/contracts
> Compiled successfully using:
   - solc: 0.8.19+commit.7dd6d404.Emscripten.clang


Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


1_initial_migrations.js
=======================

   Replacing 'UserRegistration'
   ----------------------------
   > transaction hash:    0x16a9af255c2c695e37eee99305bb2c74b089834f6b4bfef7618ffb99435ad1d4
   > Blocks: 0            Seconds: 0
   > contract address:    0x78133632F957162bdA3C0f7A472abba0DC983865
   > block number:        67
   > block timestamp:     1684739511
   > account:             0x3b46755B8DB3EE2758620f084b9dBAd65b5Ac73b
   > balance:             76.89604527567286742
   > gas used:            755889 (0xb88b1)
   > gas price:           2.500646439 gwei
   > value sent:          0 ETH
   > total cost:          0.001890211136129271 ETH
```

Constraints and Assumptions

Transparency in verifying individuals who withdraw funds from a pool is crucial in preventing money laundering. Money laundering refers to the process of making illegally obtained funds appear legal by disguising their true source. In the context of verifying fund withdrawals from a pool, transparency plays a vital role in ensuring that the funds being withdrawn are legitimate and not derived from illegal activities.

Without proper verification processes, individuals could potentially exploit the pool to launder money. They might deposit illicit funds into the pool and then attempt to withdraw them, effectively legitimizing the tainted money. This poses significant risks not only to the integrity of the pool but also to the overall financial system.

To prevent money laundering, robust verification mechanisms should be in place. These mechanisms may include Know Your Customer (KYC) procedures, which involve collecting and verifying identifying information from individuals before allowing them to access and withdraw funds from the pool. KYC processes typically require individuals to provide valid identification documents, proof of address, and other relevant information.

Additionally, transaction monitoring and suspicious activity detection tools can be implemented to flag and investigate any unusual or suspicious withdrawal patterns. Regular audits and regulatory compliance checks can further enhance the transparency and integrity of the verification process.

By implementing stringent verification procedures and maintaining a transparent system, the risk of money laundering can be mitigated, ensuring that the funds being withdrawn from the pool are legitimate and complying with legal and regulatory requirements.

Future Work

1. Social Sharing and Communication
2. Transparency and Accountability
3. Security and Privacy
4. Reporting and Analytics
5. Regulatory Compliance
6. Donor Dashboard

Reference

1. Saadat, M. N., Abdul Halim, S., Osman, H., Mohammad Nassr, R., & F. Zuhairi, M. (2019). Blockchain based crowdfunding systems. *Indonesian Journal of Electrical Engineering and Computer Science*, *15*(1), 409. https://doi.org/10.11591/ijeecs.v15.i1.pp409-413