Hanoi University of Science and Technology
School of Information and Communication Technology

PROJECT III REPORT

Decentralized Crowdfunding Application

Student Name	Lai Quang Huy
Student ID	20194438
Instructor	Prof. Hai Van Pham

Hanoi, 2023

Table of contents

I. Introduction **5**
1. Background and motivation 5
2. Objectives and scope 6
2.1. Objectives 6
2.2. Scope 6
3. Methodology 7

II. Blockchain Technology **8**
1. Overview of blockchain 8
2. Architecture of a blockchain 8
2.1. Architecture 8
2.2. Consensus Mechanisms 9
2. Key features of blockchain 9
3. Types of blockchain 10
4. Advantages and disadvantages of blockchain 11

III. Smart Contracts **12**
1. Definition and purpose of smart contracts 12
1.1. Definition of smart contracts 12
1.2. Applications 12
1.3. Purpose of smart contracts 13
2. Advantages of smart contracts 14
2.1. Advantages 14
2.2. Examples 14
3. Languages for writing smart contracts 16
4. Ethereum Smart Contract architecture 17
4.1. Architecture 17
4.2. Examples 18

IV. Solidity Programming Language **19**
1. Overview of Solidity 19
2. Data types, variables 20
3. Functions 21
3.1. Basic function modifiers 21
3.2. Custom function modifiers 22
3.3. Inheritance 24

4. Control structures 25
4.1. Conditional statements 25
4.2. Loops 25
4.3. Events 26
5. Exception handling 28
5.1. Types of exceptions 28
5.2. Try-catch statement 29
6. Abstract contracts & interfaces 30
6.1. Abstract contracts 30
6.2. Interfaces 31

V. Web3 Technology 33
1. Overview of Web3 33
2. Interaction between DApp and blockchain 34
3. Key features of Web3.js 35
4. Examples of Web3 applications 36

VI. Crowdfunding on the Blockchain 37
1. Overview of crowdfunding 37
2. Advantages of using blockchain for crowdfunding 38
3. Comparison of traditional and blockchain crowdfunding 39

VII. Design and Implementation of the Decentralized Crowdfunding App 41
1. System architecture and design 41
2. Smart contract code and implementation 42
2.1. Original Solidity code 42
2.2. Struct of Campaign 44
2.3. Create Campaign 45
2.4. Donate to Campaign 47
2.5. Get donations from Campaign 49
2.6. Get all Campaigns 50
3. User interface and user experience design 51
3.1. Frameworks 51
3.2. Overview 51
3.3. Home page 52
3.4. Browse page 53
3.5. Create page 55
3.6. Profile page 56

VIII. Testing and Deployment **57**
1. Testing the smart contract and the DApp 57
1.1. Hardhat 58
1.2. Ethereum Test Network 60
1.2. Ethers.js 62
2. Deployment of the DApp 63
2.1. ThirdWeb 63
2.2. NextJS 64

IX. Evaluation and Conclusion **65**
1. Evaluation of the decentralized crowdfunding app 65
2. Conclusion and future work 66

X. References **67**

XI. Application Demo **68**
1. Live demo 68
2. Local environment setup 68

I. Introduction

1. Background and motivation

Crowdfunding has become an increasingly popular method for raising funds for various projects and causes. However, traditional crowdfunding platforms are often centralized and may charge high fees or restrict access for certain individuals or groups. Moreover, these platforms may lack transparency and accountability, leading to concerns about fraud and misuse of funds.

Decentralized crowdfunding apps built on blockchain technology can address these issues by offering a more transparent, efficient, and secure way to raise funds. Such apps can enable anyone to contribute to projects without intermediaries, and the funds raised can be tracked and managed through smart contracts on the blockchain.

In this report, we will explore the design and implementation of a decentralized crowdfunding app on the Ethereum blockchain. The app will allow project creators to create and manage crowdfunding campaigns, and contributors to browse and donate to these campaigns using cryptocurrency.

We will also discuss the basics of blockchain technology, the Solidity programming language, and Web3 technology, which are essential for building decentralized apps on Ethereum.

The motivation for this project is to explore the potential of blockchain technology for democratizing access to funding and to provide a practical example of a decentralized app that can be used for real-world crowdfunding projects.

2. Objectives and scope

2.1. Objectives

The objectives of this project are to:

1. Design and implement a decentralized crowdfunding app on the Ethereum blockchain using Solidity and Web3 technologies.
2. Allow creators to create and manage crowdfunding campaigns, contributors to browse and donate to these campaigns using cryptocurrency.
3. Ensure transparency and accountability of funds raised by tracking and managing them through smart contracts on the blockchain.
4. Demonstrate the advantages of using blockchain technology for crowdfunding, such as increased transparency, security, and accessibility.

2.2. Scope

The scope of this project includes:

1. Developing the smart contract code for creating and managing crowdfunding campaigns.
2. Implementing a user interface for project creators to create and manage campaigns, and for contributors to browse and donate to campaigns.
3. Testing the smart contract and the user interface to ensure the app is functional and secure.
4. Deploying the app to a test network or the main Ethereum network.
5. Providing documentation and instructions for using the app.

This project will not cover:

1. Marketing and promotion of the app.
2. Legal and regulatory compliance for crowdfunding campaigns.
3. Integration with external services, such as payment gateways.
4. The focus of this project is on the technical implementation of a decentralized crowdfunding app on the Ethereum blockchain, and its potential to provide a more transparent, efficient, and accessible way to raise funds.

3. Methodology

The methodology for this project involves several stages, including:

1. Research and Analysis: This stage involves researching and analyzing the existing decentralized crowdfunding apps and the underlying blockchain technology. We will also analyze the requirements for the app and design the system architecture.
2. Smart Contract Development: This stage involves developing the smart contract code using Solidity programming language. We will implement the contract logic for creating and managing crowdfunding campaigns.
3. User Interface Development: This stage involves developing the user interface for project creators and contributors to interact with the smart contract. We will use web technologies such as HTML, CSS, and JavaScript to develop the interface.
4. Testing: This stage involves testing the smart contract and the user interface to ensure the app is functional and secure. We will conduct unit tests and integration tests using various testing frameworks.
5. Deployment: This stage involves deploying the app to a test network or the main Ethereum network. We will use the appropriate tools and protocols to deploy the smart contract and the user interface.
6. Documentation: This stage involves documenting the app and providing instructions for using it. We will prepare user manuals, technical documentation, and other relevant materials.

Throughout the project, we will follow a structured development process and use various development tools and platforms, such as Remix, Truffle, Ganache, Hardhat and Thirdweb, to facilitate the development and testing process. We will also use version control systems to manage the codebase and document the progress and results of each stage.

II. Blockchain Technology

1. Overview of blockchain

Blockchain technology is a distributed ledger technology that enables secure and transparent record-keeping of transactions. It was initially developed for use in the cryptocurrency Bitcoin, but it has since found use in various other applications, including crowdfunding, supply chain management, and identity management. In this section, we will discuss the basics of blockchain technology, including its architecture, consensus mechanisms, types, and security features.

2. Architecture of a blockchain

2.1. Architecture

The architecture of a blockchain consists of several key components, including:

Nodes: Nodes are individual computers or devices that participate in the blockchain network. Each node has a copy of the entire blockchain ledger and can verify and validate transactions.

Blocks: Blocks are collections of verified transactions that are added to the blockchain ledger. Each block contains a cryptographic hash of the previous block, forming a chain of blocks.

Consensus Mechanism: Consensus mechanism is a process that enables nodes in the network to agree on the state of the blockchain ledger. There are several consensus mechanisms, including proof of work (PoW), proof of stake (PoS), and delegated proof of stake (DPoS).

Smart Contracts: Smart contracts are self-executing contracts that contain the terms and conditions of an agreement between two or more parties. They are written in programming languages, such as Solidity, and are stored on the blockchain network.

2.2. Consensus Mechanisms

Consensus mechanisms ensure that the nodes in the network agree on the state of the blockchain ledger. The most commonly used consensus mechanisms are PoW, PoS, and DPoS.

Proof of Work (PoW): PoW is a consensus mechanism used by Bitcoin and several other cryptocurrencies. In PoW, nodes in the network compete to solve a complex mathematical puzzle. The first node to solve the puzzle is rewarded with cryptocurrency, and the block is added to the blockchain ledger.

Proof of Stake (PoS): PoS is a consensus mechanism used by several newer cryptocurrencies, including Ethereum. In PoS, nodes in the network are chosen to validate transactions based on the amount of cryptocurrency they hold. Nodes with more cryptocurrency have a higher chance of being chosen to validate transactions.

Delegated Proof of Stake (DPoS): DPoS is a consensus mechanism used by several blockchain networks, including EOS. In DPoS, token holders vote for nodes that will validate transactions. The nodes with the most votes are chosen to validate transactions.

2. Key features of blockchain

Blockchain technology offers several security features, including:

Decentralization: Decentralization means that there is no central authority controlling the network, which makes it resistant to censorship and single points of failure. For example, if one node in a blockchain network goes down, the other nodes can continue to operate without interruption. Bitcoin and Ethereum are decentralized blockchains.

Immutable Ledger: Once a transaction is added to a blockchain, it cannot be altered or deleted. This means that the data stored on the blockchain is tamper-proof and transparent. This feature is particularly useful for financial transactions, where accountability and transparency are important.

Cryptography: Blockchain uses cryptography to secure the network and protect transactions. Public key cryptography is used to authenticate users and digital signatures are used to ensure that transactions are genuine.

Consensus Mechanisms: Consensus mechanisms are used to validate transactions on the blockchain. Different consensus mechanisms have different levels of security and performance. For example, Proof of Work (PoW) is used by Bitcoin and requires miners to solve complex mathematical problems to validate transactions. Proof of Stake (PoS), on the other hand, is used by Ethereum and requires validators to hold a certain amount of cryptocurrency as collateral.

Examples of security features in blockchain technology include the use of private and public keys, digital signatures, and hashing algorithms. These security features ensure that data stored on the blockchain is safe and secure.

3. Types of blockchain

There are three types of blockchain:

Public Blockchain: Public blockchains are open to anyone and everyone can participate in the network. Anyone can read, write, and validate transactions on a public blockchain. Examples of public blockchains include Bitcoin and Ethereum.

Private Blockchain: Private blockchains are restricted to a specific group of participants. Only authorized participants can read, write, and validate transactions on a private blockchain. Examples of private blockchains include Hyperledger Fabric and Corda.

Consortium Blockchain: Consortium blockchains are semi-private blockchains where the consensus process is controlled by a pre-selected group of nodes. Consortium blockchains are used by businesses and organizations to collaborate and share information securely. Examples of consortium blockchains include R3 Corda and Quorum.

4. Advantages and disadvantages of blockchain

Blockchain technology offers several advantages and disadvantages. Some of the key advantages include:

1. Decentralization: Blockchain technology is decentralized, which means that it is not controlled by any single entity. This makes it more resistant to censorship and hacking.
2. Transparency: Transactions on the blockchain are transparent and can be viewed by anyone. This makes it easier to verify transactions and prevent fraud.
3. Security: Blockchain technology uses cryptographic algorithms to secure the network and protect transactions.
4. Efficiency: Blockchain technology is faster and more efficient than traditional financial systems because it eliminates the need for intermediaries.
5. Accessibility: Blockchain technology is accessible to anyone with an internet connection, which makes it easier for people to participate in the economy.

However, blockchain technology also has several disadvantages, including:

1. Scalability: Blockchain technology can be slow and inefficient when it comes to processing large amounts of data.
2. Energy consumption: Some blockchain networks, such as Bitcoin, consume a large amount of energy to validate transactions.
3. Regulation: Because blockchain technology is decentralized, it can be difficult to regulate.
4. Complexity: Blockchain technology is complex and can be difficult for the average person to understand.

Overall, while blockchain technology has many potential benefits, it is not without its drawbacks.

III. Smart Contracts

1. Definition and purpose of smart contracts

1.1. Definition of smart contracts

A smart contract is a self-executing program that can be encoded onto a blockchain. It contains a set of rules and conditions that automatically execute when certain predefined conditions are met. Smart contracts are designed to enable transparent, secure, and efficient transactions between parties without the need for intermediaries.

Comparison with traditional contracts: Traditional contracts are usually written in natural language and require intermediaries, such as lawyers or banks, to enforce the terms of the contract. Smart contracts, on the other hand, are self-executing and enforceable through code. They eliminate the need for intermediaries, which reduces costs and increases efficiency.

Smart contracts are an integral part of blockchain development because they enable the execution of decentralized applications (dApps) on the blockchain. By using smart contracts, developers can create self-executing programs that run on the blockchain and interact with other applications and users. Smart contracts are executed on the Ethereum Virtual Machine (EVM), which is a decentralized, Turing-complete virtual machine that runs on the Ethereum blockchain.

1.2. Applications

Smart contracts can be used in a variety of applications, such as:

Supply chain management: Smart contracts can be used to track the movement of goods from production to distribution. They can be programmed to trigger payments or transfers of ownership automatically when certain conditions are met, such as delivery of goods or confirmation of receipt.

Financial services: Smart contracts can be used to automate financial transactions, such as payments, loans, and insurance. They can be programmed to execute payments automatically when certain conditions are met, such as the completion of a service or the occurrence of a specific event.

Real estate transactions: Smart contracts can be used to automate the transfer of ownership of real estate properties. They can be programmed to transfer ownership automatically when certain conditions are met, such as the payment of the purchase price or the registration of the transfer with the relevant authorities.

1.3. Purpose of smart contracts

The purpose of smart contracts is to automate the execution of agreements and transactions between parties. Smart contracts eliminate the need for intermediaries and reduce the risk of fraud, errors, and delays. They can be used for a wide range of applications, such as financial services, supply chain management, real estate transactions, and more.

2. Advantages of smart contracts

2.1. Advantages

Transparency: Smart contracts are transparent because they are stored on the blockchain, which is a distributed ledger that is accessible to all participants. This means that all parties can see the terms and conditions of the contract and the outcome of the transaction.

Security: Smart contracts are secure because they are stored on the blockchain, which is tamper-proof and resistant to hacking. Smart contracts are also encrypted and can only be accessed by authorized parties.

Efficiency: Smart contracts are efficient because they automate the execution of agreements and transactions, eliminating the need for intermediaries and reducing the time and cost of transactions.

Accuracy: Smart contracts are accurate because they are executed automatically and are not subject to human error. This reduces the risk of mistakes and discrepancies.

Trust: Smart contracts enable trust between parties because they are executed automatically and cannot be changed without the agreement of all parties. This eliminates the need for trust in intermediaries, such as banks or lawyers, and reduces the risk of fraud.

2.2. Examples

Examples of applications that benefit from the advantages of smart contracts include:

Decentralized finance (DeFi): Smart contracts are used in DeFi applications to automate financial transactions, such as lending, borrowing, and trading. They enable transparent and secure transactions without the need for intermediaries.

Supply chain management: Smart contracts are used in supply chain management to automate the tracking and transfer of goods. They enable transparency and efficiency in the supply chain by automating the execution of agreements and transactions.

Real estate transactions: Smart contracts are used in real estate transactions to automate the transfer of ownership of properties. They enable transparency, security, and efficiency in the transfer of assets.

3. Languages for writing smart contracts

Solidity: Solidity is a high-level programming language that is used to write smart contracts on the Ethereum blockchain. It is similar to JavaScript and has a syntax that is easy to learn. Solidity is the most popular language for writing smart contracts and has a large community of developers.

Vyper: Vyper is a programming language that is similar to Python and is used to write smart contracts on the Ethereum blockchain. It is designed to be more secure and easier to audit than Solidity, but it has a smaller community of developers.

Serpent: Serpent is a programming language that is similar to Python and was one of the first languages used to write smart contracts on the Ethereum blockchain. It is no longer actively maintained and has been replaced by Solidity and Vyper.

Other languages: While Solidity, Vyper, and Serpent are the most commonly used languages for writing smart contracts on the Ethereum blockchain, other languages such as C++, Rust, and JavaScript can also be used.

Examples of smart contract applications written in different languages include:

- A Solidity-based smart contract for a decentralized exchange (DEX) on the Ethereum blockchain
- A Vyper-based smart contract for a prediction market on the Ethereum blockchain
- A C++-based smart contract for a supply chain management system on the EOS blockchain

4. Ethereum Smart Contract architecture

4.1. Architecture

Account-based Model: The account-based model is used by the Ethereum blockchain to store and manage smart contracts. Each account on the Ethereum blockchain has a balance and a unique address, and can be either an externally-owned account (EOA) or a contract account.

Contract Account: A contract account is a special type of account on the Ethereum blockchain that is used to store and execute smart contracts. When a contract is deployed on the Ethereum blockchain, a contract account is created with a unique address. This contract account is then used to store the code and data of the smart contract, and execute its functions.

Gas: Gas is a unit of measurement for the amount of computational power required to execute a function on the Ethereum blockchain. Each function in a smart contract requires a certain amount of gas to execute, and users must pay a fee in Ether (ETH) to cover the cost of the gas. The amount of gas required for a function depends on the complexity of the function and the amount of data it processes.

EVM: The Ethereum Virtual Machine (EVM) is the runtime environment for executing smart contracts on the Ethereum blockchain. The EVM is a virtual machine that is designed to be deterministic and secure, and ensures that smart contracts execute exactly as they are written.

Interacting with Smart Contracts: Smart contracts on the Ethereum blockchain can be interacted with using a web3 provider such as MetaMask, which allows users to send transactions to the smart contract and execute its functions. Smart contracts can also interact with other smart contracts on the Ethereum blockchain using their unique addresses.

4.2. Examples

Examples of smart contract applications built on the Ethereum blockchain include:

- A decentralized autonomous organization (DAO) that allows members to vote on proposals and make decisions using a smart contract
- A peer-to-peer lending platform that allows borrowers and lenders to transact using a smart contract on the Ethereum blockchain
- A supply chain management system that uses a smart contract to track the movement of goods and ensure authenticity and traceability

IV. Solidity Programming Language

1. Overview of Solidity

Solidity is a high-level object-oriented programming language used to write smart contracts on the Ethereum blockchain. It is a statically-typed language, which means that variables have to be declared with a specific data type before they can be used. Solidity is influenced by C++, Python, and JavaScript, and its syntax is similar to these languages.

Smart contracts written in Solidity can be executed on the Ethereum Virtual Machine (EVM), which is a decentralized computing platform that runs on the Ethereum blockchain. Solidity supports a wide range of data types, including integers, booleans, strings, arrays, and mappings. It also supports inheritance, libraries, and interfaces, which allow for code reuse and modularity.

Solidity contracts can interact with other contracts on the Ethereum blockchain, as well as with external applications through APIs. Solidity is also designed to be secure, and it includes features such as function modifiers, access control, and exception handling to prevent security vulnerabilities.

Solidity code is compiled into bytecode, which is executed on the EVM. Solidity is the most popular programming language for writing smart contracts on the Ethereum blockchain.

2. Data types, variables

Solidity has a variety of built-in data types that can be used to define variables, including:

- Bool: boolean values true or false
- Int and uint: signed and unsigned integers of various sizes (int8, uint8, int256, uint256, etc.)
- Address: a 20-byte Ethereum address
- String: dynamic length string of characters
- Bytes and byte arrays: static or dynamic length sequences of bytes
- Variables in Solidity are defined with the syntax:

```
<Type> <variable_name>;
```

For example, to define a uint variable called myVariable:

```
uint myVariable;
```

3. Functions

Functions in Solidity are defined with the function keyword, followed by the function name and any parameters in parentheses, and then the function body enclosed in curly braces. For example:

```
function addNumbers(uint x, uint y) public returns (uint) {
    uint result = x + y;
    return result;
}
```

In this example, the addNumbers function takes two uint parameters x and y, adds them together, and returns the result as a uint.

3.1. Basic function modifiers

Solidity also supports function modifiers, which are used to change the behavior of a function. For example, the view modifier can be used to indicate that a function does not modify state, and the pure modifier can be used to indicate that a function does not read or modify state.

1. Pure modifier: This modifier is used for functions that do not read from or modify the state of the contract. Here's an example:

```
function add(uint256 x, uint256 y) public pure returns (uint256)
{
    return x + y;
}
```

In this example, the add function takes two unsigned integers as inputs, adds them together, and returns the result. Because the function does not modify the state of the contract, it is marked as pure.

2. View modifier: This modifier is used for functions that do not modify the state of the contract, but do read from it. Here's an example:

```
function getBalance(address _address) public view returns
(uint256) {
    return balances[_address];
}
```

In this example, the getBalance function takes an Ethereum address as input, reads the value of the balances mapping for that address, and returns the result. Because the function does not modify the state of the contract, but does read from it, it is marked as view.

3.2. Custom function modifiers

1. OnlyOwner: This modifier can be used to restrict the execution of a function to the owner of the contract.

```
address public owner;

modifier onlyOwner {
    require(msg.sender == owner, "Only the contract owner can
call this function.");
    _;
}
function doSomething() public onlyOwner {
    // execute if the caller is the owner
}
```

2. RestrictValue: This modifier can be used to restrict the value that can be passed as a parameter to a function.

```
uint public maxValue = 100;

modifier restrictValue(uint _value) {
    require(_value ⩽ maxValue, "Value exceeds the maximum
allowed.");
    _;
}
```

```
function setValue(uint _newValue) public
restrictValue(_newValue) {
    // execute if the value is within the limit
}
```

3. CheckTime: This modifier can be used to restrict the execution of a function to a certain time interval. For example:

```
uint public startTime;
uint public endTime;

modifier checkTime {
    require(block.timestamp ⩾ startTime && block.timestamp ⩽
endTime, "Function can only be called during a certain time
interval.");
    _;
}
function doSomething() public checkTime {
    // execute if the current time is within the specified
interval
}
```

4. Authenticated: This modifier can be used to restrict the execution of a function to a user who has been authenticated by the contract. For example:

```
mapping (address ⇒ bool) public authenticatedUsers;

modifier authenticated {
    require(authenticatedUsers[msg.sender], "User is not
authenticated.");
    _;
}
function doSomething() public authenticated {
    // execute if the user has been authenticated
}
```

3.3. Inheritance

Inheritance is an important feature of Solidity that allows you to reuse code and create more complex contracts. Inheritance allows you to create new contracts that inherit properties and functions from existing contracts. This can be useful for creating modular and reusable code.

In Solidity, inheritance is implemented using the keyword `is`. A contract that inherits from another contract is called a derived contract, while the contract it inherits from is called a base contract. The derived contract inherits all the functions and variables from the base contract and can add its own functions and variables.

Here's an example of a simple base contract and a derived contract:

```
// Base contract
contract MyBaseContract {
    uint myVariable;
    function setMyVariable(uint _value) public {
        myVariable = _value;
    }
}
// Derived contract that inherits from MyBaseContract
contract MyDerivedContract is MyBaseContract {
    function getMyVariable() public view returns (uint) {
        return myVariable;
    }
}
```

In this example, MyDerivedContract inherits the myVariable variable and the setMyVariable function from MyBaseContract. It also adds its own function getMyVariable, which returns the value of myVariable.

Inheritance is useful because it allows us to create more complex contracts by building on top of existing ones. It also helps to make our code more modular and easier to maintain.

4. Control structures

Solidity supports a variety of control structures and functions that allow for complex logic and decision-making within smart contracts.

By using these control structures and functions, developers can create complex smart contracts that implement a wide range of functionality.

Solidity also supports a number of other features, including inheritance, interfaces, and libraries, which allow for even greater flexibility and modularity in smart contract design.

4.1. Conditional statements

Solidity supports the use of if, else if, and else statements for implementing conditional logic within smart contracts. These statements can be used to control the flow of execution based on specific conditions.

```
function checkAge(uint age) public pure returns (string memory)
{
    if (age < 18) {
        return "You are not old enough to vote.";
    } else {
        return "You can vote in the election.";
    }
}
```

4.2. Loops

Solidity supports several types of loops including for, while, and do-while loops. These loops can be used to execute a block of code repeatedly until a certain condition is met.

```
function count(uint start, uint end) public pure returns (uint)
{
    uint total = 0;
    for (uint i = start; i ≤ end; i++) {
        total += i;
    }
    return total;
}

function double(uint num) public pure returns (uint) {
    uint result = num;
    while (result < 100) {
        result *= 2;
    }
    return result;
}
```

4.3. Events

Events are another important feature of Solidity, and they allow smart contracts to emit notifications when certain conditions are met. Events are defined using the event keyword, and they can be subscribed to by external applications to receive notifications about the state of the smart contract. Events allow smart contract developers to notify the user interface of their DApp about changes in the state of the smart contract.

```
contract EventExample {
    event NewMessage(string message);
    string public message;

    function setMessage(string memory newMessage) public {
        message = newMessage;
        emit NewMessage(newMessage);
    }
}
```

We have a contract named EventExample. The contract has a public string variable named message and a function named setMessage that allows us to set the value of the message variable.

We have also defined an event named NewMessage that takes a string parameter. This event will be triggered whenever we call the setMessage function.

The emit keyword is used to trigger an event in Solidity. In this case, we are emitting the NewMessage event and passing it the new message that was set.

5. Exception handling

5.1. Types of exceptions

There are two types of exceptions in Solidity: require exceptions and assert exceptions.

1. Require Exception

The require exception is used to check that certain conditions are met before executing a function. If the condition is not met, then the function will throw an exception and revert any changes made in the transaction. Here's an example:

```
function buyTokens(uint256 amount) public payable {
    require(msg.value ⩾ amount * tokenPrice, "Insufficient
ether provided.");
    // code to transfer tokens to the buyer
}
```

The function buyTokens checks that the amount of ether sent by the user is sufficient to purchase the requested number of tokens. If the condition is not met, the function will throw an exception with the message "Insufficient ether provided."

2. Assertion Exception

The assertion exception is used to check for internal errors in the code. If an assertion fails, it means that there is a bug in the code and the transaction will be reverted. Here's an example:

```
function withdraw() public {
    assert(balances[msg.sender] > 0);
    uint256 amount = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(amount);
```

```
}
```

The function withdraw checks that the user has a positive balance before allowing them to withdraw their funds. The assert statement checks for internal errors, such as a negative balance, and if it fails, the transaction will be reverted.

Revert exceptions occur when a function encounters an error condition and needs to revert the changes made to the state of the contract. For example, a function that transfers tokens from one account to another may encounter a revert exception if the sender does not have enough tokens to complete the transfer.

Assert exceptions, on the other hand, are used to validate certain conditions that are expected to be true. If an assert statement evaluates to false, the transaction is immediately reverted and all changes to the contract state are undone.

5.2. Try-catch statement

To handle exceptions in Solidity, you can use the try-catch statement. The try block contains the code that may throw an exception, while the catch block contains the code that handles the exception.

```
function transferTokens(address recipient, uint amount) public {
    try token.transfer(recipient, amount) {
        emit TransferSuccessful(recipient, amount);
    } catch {
        emit TransferFailed(recipient, amount);
    }
}
```

The transferTokens function attempts to transfer amount tokens to the recipient address using the transfer function of the token contract. If the transfer is successful, the TransferSuccessful event is emitted. Otherwise, the TransferFailed event is emitted. The catch block handles any exceptions that may be thrown by the transfer function.

Overall, exception handling is an important feature of Solidity that helps ensure the robustness and reliability of smart contracts.

6. Abstract contracts & interfaces

Both abstract contracts and interfaces allow for contracts to implement a set of standardized functions and events, which can be used by other contracts and applications. By defining a set of standard functions and events, we can create interoperability between different contracts and systems, allowing them to work together seamlessly.

6.1. Abstract contracts

```
abstract contract Payment {
    function pay(uint amount) public virtual;
}
```

An abstract contract is a special type of contract in Solidity that cannot be instantiated on its own. Instead, it is designed to be inherited by other contracts.

Abstract contracts contain abstract functions that do not have an implementation. These functions are meant to be implemented by the child contracts that inherit the abstract contract.

An abstract contract is defined using the abstract keyword, and abstract functions are defined using the function keyword with no implementation.

```
abstract contract Token {
    function balanceOf(address account) public virtual view
returns (uint256);
    function transfer(address recipient, uint256 amount) public
virtual returns (bool);
    function approve(address spender, uint256 amount) public
virtual returns (bool);
    function allowance(address owner, address spender) public
virtual view returns (uint256);
    function transferFrom(address sender, address recipient,
uint256 amount) public virtual returns (bool);
```

```
    function increaseAllowance(address spender, uint256
addedValue) public virtual returns (bool);
    function decreaseAllowance(address spender, uint256
subtractedValue) public virtual returns (bool);
}
```

In this example, we define an abstract contract called Token. The contract defines a set of functions that are required for any token contract.

The virtual keyword indicates that these functions are not implemented in the abstract contract and must be implemented in derived contracts. This allows us to define a set of standard functions that can be used by any token contract, while allowing each contract to define its own implementation.

6.2. Interfaces

```
interface Token {
   function transfer(address to, uint amount) external returns (bool);
   function balanceOf(address owner) external view returns (uint);
}
```

An interface is similar to an abstract contract, but it is even more abstract. An interface is a contract that defines a set of functions and events that another contract can implement.

Unlike abstract contracts, interfaces cannot have any state variables or function implementations. Interfaces are used to define a standard set of functions that can be implemented by different contracts, making it easier to interact with them.

An interface is defined using the interface keyword, and functions are defined using the function keyword with no implementation.

The above Token interface defines two functions: transfer() and balanceOf(). Any contract that wants to interact with a token that implements this interface can use these functions, regardless of the specific implementation of the token contract.

```
interface ERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns
(uint256);
    function transfer(address recipient, uint256 amount)
external returns (bool);
    function allowance(address owner, address spender) external
view returns (uint256);
    function approve(address spender, uint256 amount) external
returns (bool);
    function transferFrom(address sender, address recipient,
uint256 amount) external returns (bool);
    event Transfer(address indexed from, address indexed to,
uint256 value);
    event Approval(address indexed owner, address indexed
spender, uint256 value);
}
```

In this example, we define an interface called ERC20. The interface specifies a set of functions that a contract must implement in order to be considered an ERC20 token.

In addition to the functions, the interface also specifies two events, Transfer and Approval, that are emitted by ERC20 tokens when tokens are transferred or approvals are granted.

V. Web3 Technology

1. Overview of Web3

Web3 is a set of technologies that enables developers to build decentralized applications (DApps) that can interact with blockchain networks. It provides a way to communicate with the blockchain network and its smart contracts, allowing users to create, manage, and transfer digital assets in a secure and decentralized manner.

Web3 technology comprises several components, including Web3.js, Web3.py, and Web3j, which are libraries for interacting with Ethereum using JavaScript, Python, and Java, respectively. These libraries enable developers to create DApps that can communicate with Ethereum nodes, smart contracts, and the Ethereum Virtual Machine (EVM).

Web3 technology also includes protocols and standards such as the Ethereum Request for Comments (ERC) series, which define standards for token issuance and smart contract development, as well as the InterPlanetary File System (IPFS) and Whisper, which are decentralized storage and messaging protocols, respectively.

Web3 technology has enabled the development of a range of DApps, including decentralized exchanges, prediction markets, games, and crowdfunding platforms. These applications are built on top of the Ethereum blockchain and leverage its decentralized nature to provide users with a secure and transparent environment for conducting transactions and managing digital assets.

2. Interaction between DApp and blockchain

A Decentralized Application (DApp) is a computer application that runs on a blockchain network. DApps interact with the blockchain through smart contracts, which are self-executing contracts with the terms of the agreement directly written into code. When a DApp interacts with the blockchain, it sends a transaction to a smart contract, which executes the transaction and updates the state of the blockchain.

Web3.js is a JavaScript library that provides an API for interacting with the Ethereum blockchain. It allows developers to write DApps that can interact with the blockchain, such as sending transactions to smart contracts, retrieving data from the blockchain, and listening for events.

To interact with the blockchain, a DApp needs a connection to an Ethereum node. An Ethereum node is a computer that runs the Ethereum software and maintains a copy of the entire blockchain. There are two types of nodes: full nodes and light nodes. Full nodes store the entire blockchain and validate every transaction on the network. Light nodes only store block headers and can only validate transactions related to their accounts.

Web3.js provides a way to connect to an Ethereum node using the HTTP, WebSocket, or IPC protocol. Once connected, a DApp can use the Web3.js API to interact with the blockchain.

In summary, the interaction between a DApp and the blockchain is through smart contracts, and Web3.js provides an API for developers to interact with the blockchain. To connect to the blockchain, a DApp needs a connection to an Ethereum node.

3. Key features of Web3.js

Web3.js is a library for interacting with Ethereum blockchain nodes. It provides a simple API for developers to interact with the blockchain using JavaScript. Some of the key features of Web3.js include:

1. Contract Abstraction: Web3.js provides a simple and intuitive way to interact with smart contracts on the Ethereum blockchain. It includes a contract abstraction layer that allows developers to interact with smart contracts as if they were JavaScript objects.
2. Web3 Providers: Web3.js supports various Ethereum providers such as MetaMask, Infura, and Geth. Providers are essential for connecting to the Ethereum network and accessing blockchain data.
3. Events: Web3.js provides an event system that allows developers to listen to blockchain events in real-time. Events can be triggered by smart contracts, transactions, and other blockchain activities.
4. Promises: Web3.js uses promises to handle asynchronous operations, making it easy for developers to write code that interacts with the blockchain.
5. Decentralized Applications: Web3.js is a critical tool for building decentralized applications (DApps) on the Ethereum blockchain. It provides developers with the tools they need to build front-end interfaces that interact with smart contracts on the blockchain.
6. User Management: Web3.js provides a simple way to manage user accounts and wallets. Developers can use Web3.js to create new accounts, sign transactions, and interact with the blockchain securely.
7. Support for Multiple Languages: Web3.js supports multiple programming languages, including JavaScript, Python, and Java.
8. Interoperability: Web3.js is designed to work with other Ethereum tools and technologies, such as Solidity, Truffle, and Remix. This interoperability makes it easy for developers to integrate Web3.js into their existing development workflows.

Web3.js is a powerful tool for developers building decentralized applications on the Ethereum blockchain. Its intuitive API and support for multiple languages make it an accessible tool for developers of all skill levels. Its support for contract abstraction, events, and promises make it easy to interact with the blockchain and build front-end interfaces that interact with smart contracts.

4. Examples of Web3 applications

1. CryptoKitties

CryptoKitties is a blockchain-based game where players can buy, sell, and breed virtual cats.

2. MakerDAO

MakerDAO is a decentralized lending platform where users can borrow and lend cryptocurrency using smart contracts.

3. Augur

Augur is a decentralized prediction market platform where users can make predictions on real-world events.

4. Golem

Golem is a decentralized computing platform that allows users to rent out their unused computing power for tasks like rendering graphics.

5. Uniswap

Uniswap is a decentralized exchange where users can trade cryptocurrencies without a central authority.

Each of these applications demonstrates different use cases for Web3 technology and how it can be used to create decentralized platforms and services.

VI. Crowdfunding on the Blockchain

1. Overview of crowdfunding

Crowdfunding is a process of raising funds from a large number of people (the crowd) for a particular project or initiative. This method of fundraising has become increasingly popular in recent years, especially with the rise of internet and social media. Crowdfunding can be used for various purposes such as supporting a startup, funding a creative project, or even raising funds for a charitable cause.

There are generally three types of crowdfunding:

1. Reward-based crowdfunding: where backers receive a reward or product in exchange for their contribution.
2. Equity-based crowdfunding: where backers receive a share of ownership in the company or project they are backing.
3. Donation-based crowdfunding: where backers contribute without receiving any material benefit or ownership.

Crowdfunding has many benefits such as helping entrepreneurs and innovators to access capital, providing a platform for community engagement, and enabling individuals to support causes they care about.

2. Advantages of using blockchain for crowdfunding

Increased transparency: Blockchain technology allows for a high level of transparency and accountability in the crowdfunding process. Every transaction is recorded on the blockchain and can be viewed by anyone, ensuring that the funds are used as intended.

Reduced fraud: The use of blockchain technology in crowdfunding reduces the risk of fraud and mismanagement of funds. The smart contracts used to govern the crowdfunding process are self-executing, meaning that the funds are automatically distributed to the project when certain conditions are met.

Decentralization: Blockchain technology enables the crowdfunding process to be decentralized, which means that it is not controlled by any single entity. This reduces the risk of corruption and ensures that the decision-making process is fair and impartial.

Lower costs: Traditional crowdfunding platforms charge high fees for their services, which can eat into the funds raised by the project. With blockchain crowdfunding, there are no intermediaries involved, which means that the costs are significantly reduced.

Global reach: Blockchain technology enables crowdfunding to have a global reach, which means that projects can reach a wider audience and raise more funds. This is particularly beneficial for projects that have a niche audience or are looking for funding from specific geographic regions.

3. Comparison of traditional and blockchain crowdfunding

Crowdfunding has been a popular way for individuals and organizations to raise funds for their projects, products, and services.

Traditional crowdfunding platforms like Kickstarter, Indiegogo, and GoFundMe have been used by millions of people to raise billions of dollars.

However, blockchain-based crowdfunding platforms have emerged as a new alternative that provides unique advantages over traditional crowdfunding.

Here are some of the key differences between traditional and blockchain-based crowdfunding:

1. Decentralization: Traditional crowdfunding platforms are centralized, which means that they are controlled by a single entity that decides the rules and regulations. Blockchain-based crowdfunding platforms, on the other hand, are decentralized, which means that there is no single entity controlling the platform. Instead, the platform is governed by a network of users who collectively decide the rules and regulations.
2. Transparency: Traditional crowdfunding platforms often lack transparency, which makes it difficult for backers to know how their funds are being used. Blockchain-based crowdfunding platforms, on the other hand, provide complete transparency by recording every transaction on a public blockchain. This means that backers can easily track their funds and ensure that they are being used as intended.
3. Security: Traditional crowdfunding platforms are often vulnerable to hacks and security breaches, which can result in the loss of funds. Blockchain-based crowdfunding platforms, on the other hand, use sophisticated cryptography to secure transactions and prevent fraud. This makes blockchain-based crowdfunding platforms much more secure than traditional platforms.
4. Access: Traditional crowdfunding platforms are often restricted by geography and other factors. Blockchain-based crowdfunding platforms, on the other hand, are accessible to anyone with an internet connection. This means that blockchain-based crowdfunding platforms have the potential to reach a much larger audience than traditional platforms.

5. Fees: Traditional crowdfunding platforms often charge high fees for their services, which can eat into the funds raised by the project. Blockchain-based crowdfunding platforms, on the other hand, often have lower fees or no fees at all. This means that more of the funds raised go directly to the project.

Overall, blockchain-based crowdfunding platforms offer several advantages over traditional platforms. They provide increased transparency, security, and accessibility, while also reducing fees and eliminating the need for a centralized authority.

VII. Design and Implementation of the Decentralized Crowdfunding App

1. System architecture and design

In this section, we will discuss the architecture and design of the decentralized crowdfunding app. The app is built on top of the Ethereum blockchain and utilizes smart contracts for fundraising.

The app has two main functions: creating a campaign and donating to existing campaigns.

The system architecture of the app consists of three layers: the client layer, the application layer, and the blockchain layer.

Client layer: This layer includes the user interface of the app that is used by the users to interact with the app. The client layer is implemented using web technologies such as HTML, CSS, and JavaScript. On the frontend, we used NextJS to build a modern and responsive user interface. The UI provides a simple and intuitive way for users to create new campaigns, view existing campaigns, and make donations.

Application layer: This layer includes the backend logic of the app that handles the user requests and interacts with the blockchain layer. We utilized ThirdWeb for the application layer. ThirdWeb serves as the intermediary between the frontend and the smart contract running on the blockchain. This layer handles user authentication, campaign creation, and donation processing.

Blockchain layer: This layer includes the smart contracts that are deployed on the Ethereum blockchain. The smart contracts handle the fundraising logic of the app and store the fundraising data.

Overall, the system architecture and design of the decentralized crowdfunding app are designed to provide a secure and transparent way for fundraising on the blockchain.

2. Smart contract code and implementation

2.1. Original Solidity code

```
contract CrowdFunding {
    struct Campaign {
        address owner;
        string title;
        string description;
        uint256 target;
        uint256 deadline;
        uint256 amountCollected;
        string image;
        address[] donators;
        uint256[] donations;
    }
    mapping(uint256 => Campaign) public campaigns;
    uint256 public camNum = 0;

    function create(
        address _owner,
        string memory _title,
        string memory _description,
        uint256 _target,
        uint256 _deadline,
        string memory _image
    ) public returns (uint256) {
        Campaign storage campaign = campaigns[camNum];
        require(
            _deadline > block.timestamp,
            "Deadline should be date in the future."
        );
        campaign.owner = _owner;
        campaign.title = _title;
```

```
        campaign.description = _description;
        campaign.target = _target;
        campaign.deadline = _deadline;
        campaign.image = _image;
        campaign.amountCollected = 0;
        camNum++;
        return camNum - 1;
    }
    function donate(uint256 _id) public payable {
        uint256 amount = msg.value;
        Campaign storage campaign = campaigns[_id];
        campaign.donators.push(msg.sender);
        campaign.donations.push(amount);
        (bool sent, ) = payable(campaign.owner).call{value:
amount}("");
        if (sent) {
            campaign.amountCollected = campaign.amountCollected
+ amount;
        }
    }
    function getDonators(uint256 _id) public view returns
(address[] memory, uint256[] memory) {
        return (campaigns[_id].donators,
campaigns[_id].donations);
    }
    function getCampaigns() public view returns (Campaign[]
memory) {
        Campaign[] memory allCam = new Campaign[](camNum);
        for (uint i = 0; i < camNum; i++) {
            Campaign storage item = campaigns[i];
            allCam[i] = item;
        }
        return allCam;
    }
}
```

The smart contract is written in Solidity, which is a programming language specifically designed for writing smart contracts on the Ethereum blockchain.

The smart contract code is a set of instructions that define the rules and logic of the crowdfunding app. The code is executed by the Ethereum Virtual Machine (EVM), which is a decentralized, tamper-proof runtime environment for smart contracts.

2.2. Struct of Campaign

The first part of the smart contract defines a struct called "Campaign". This struct contains the details of each crowdfunding campaign, such as the owner, title, description, target amount, deadline, amount collected, image, and arrays of donators and donations.

```
    struct Campaign {
        address owner;
        string title;
        string description;
        uint256 target;
        uint256 deadline;
        uint256 amountCollected;
        string image;
        address[] donators;
        uint256[] donations;
    }
```

The mapping "campaigns" is used to map a unique ID to each campaign struct.

```
    mapping(uint256 ⇒ Campaign) public campaigns;
```

The "camNum" variable is used to keep track of the number of campaigns created so far. This is used to generate a unique ID for each new campaign created.

```
    uint256 public camNum = 0;
```

2.3. Create Campaign

The "create" function is used to create a new crowdfunding campaign. The function takes in the campaign owner's address, title, description, target amount, deadline, and image as arguments.

The function then creates a new campaign struct, assigns the arguments to the corresponding fields, sets the amount collected to 0, and returns the unique ID of the new campaign.

Solidity code:

```
function create(
    address _owner,
    string memory _title,
    string memory _description,
    uint256 _target,
    uint256 _deadline,
    string memory _image
) public returns (uint256) {

    Campaign storage campaign = campaigns[camNum];
    require(
        _deadline > block.timestamp,
        "Deadline should be date in the future."
    );
    campaign.owner = _owner;
    campaign.title = _title;
    campaign.description = _description;
    campaign.target = _target;
    campaign.deadline = _deadline;
    campaign.image = _image;
    campaign.amountCollected = 0;
    camNum++;
    return camNum - 1;
}
```

Pseudo code:

```
INPUT: owner, title, description, target, deadline, image
OUTPUT: New campaign created

     FUNCTION create
          IF deadline > current time
               A new campaign is created
               Total number of campaign += 1
```

2.4. Donate to Campaign

The "donate" function is used to allow users to donate to a specific campaign. The function takes in the ID of the campaign and the donation amount as arguments. The function then adds the donator's address and donation amount to the respective arrays in the campaign struct.

The function then sends the donation amount to the campaign owner's address and updates the amount collected for the campaign.

Solidity code:

```
    function donate(uint256 _id) public payable {
        uint256 amount = msg.value;
        Campaign storage campaign = campaigns[_id];

        campaign.donators.push(msg.sender);
        campaign.donations.push(amount);

        (bool sent, ) = payable(campaign.owner).call{value:
amount}("");
        if (sent) {
            campaign.amountCollected = campaign.amountCollected
+ amount;
        }
    }
```

Pseudo code:

```
INPUT: campaign_id, sender, value
OUTPUT: the campaign is donated with a value from sender

     FUNCTION donate
          Add sender to list of donators of the campaign
          Add value to list of donations of the campaign
          Add value to amount collected of the campaign
```

The donate function is responsible for accepting donations from users to a specific campaign identified by its _id. The function is declared as public payable, which means it can receive Ether and allow the transaction to modify the contract state.

When the user sends Ether to the donate function, the amount is assigned to the amount variable. Next, the function accesses the campaign object associated with the given _id using the campaigns[_id] syntax.

The payable keyword enables the function to receive Ether from the transaction. It allows the contract to accept Ether in a secure and straightforward manner.

The call function is used to send Ether to the campaign owner's address by calling the payable function with an empty parameter. This is where the Ether is transferred to the campaign owner.

The if statement checks whether the transfer was successful or not. If the transfer is successful, then the campaign.amountCollected is updated to reflect the amount collected for the campaign.

In summary, the donate function receives Ether from the user, then forwards the Ether to the campaign owner's address by calling the payable function with an empty parameter. Finally, the amount collected is recorded in the corresponding campaign object.

2.5. Get donations from Campaign

The "getDonators" function is used to get the list of donators and their corresponding donations for a specific campaign. The function takes in the ID of the campaign as an argument and returns two arrays - one for the donators and one for the corresponding donations.

Solidity code:

```
    function getDonators(
        uint256 _id
    ) public view returns (address[] memory, uint256[] memory) {
        return (campaigns[_id].donators,
campaigns[_id].donations);
    }
```

Pseudo code:

```
INPUT: campaign_id
OUTPUT: list of donators and their corresponding donations

     FUNCTION getDonators
```

2.6. Get all Campaigns

The "getCampaigns" function is used to get a list of all the campaigns created so far. The function returns an array of campaign structs containing all the details of each campaign.

Solidity code:

```
    function getCampaigns() public view returns (Campaign[]
memory) {
        Campaign[] memory allCam = new Campaign[](camNum);
        for (uint i = 0; i < camNum; i++) {
            Campaign storage item = campaigns[i];
            allCam[i] = item;
        }
        return allCam;
    }
```

Pseudo code:

```
INPUT: no input
OUTPUT: list of every campaign available

     FUNCTION getCampaigns
          INITIATE an empty list that have size equal total
          numbers of campaign
          FOR each campaigns, assign campaign by their
          corresponding index
```

Overall, the smart contract code provides a simple and efficient way to create, donate, and retrieve data for crowdfunding campaigns on the blockchain.

3. User interface and user experience design

3.1. Frameworks

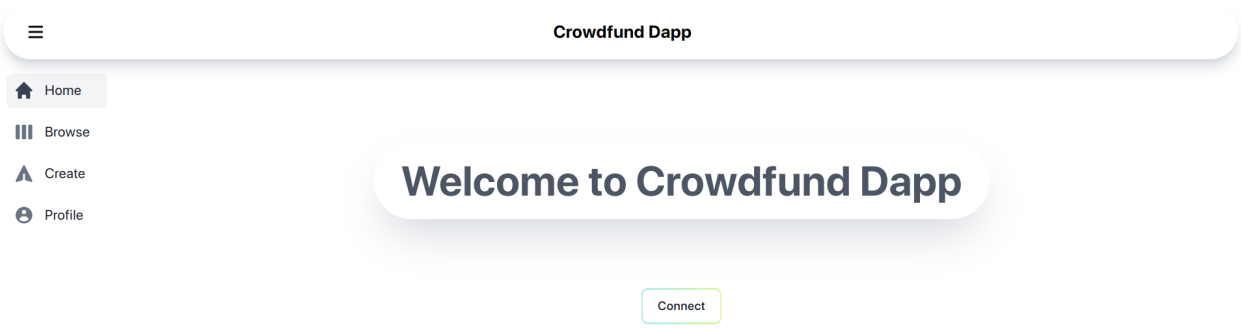
The app is designed to have a user-friendly interface with easy navigation. It is built with NextJS 13, a popular React-based framework for building web applications, and uses ThirdWeb for the application layer.

3.2. Overview

It consists of four main pages accessible via a sidebar navigation menu, each with different components and features. The sidebar is always visible, providing easy access to the different pages of the app. The four main pages are Home, Browse, Create, and Profile.

The Browse page has a modal that displays more details about a campaign and allows users to donate, while the Create page has a form that allows users to create their own campaign. The Profile page functions similarly to the Browse page but filters only campaigns owned by the connected account.

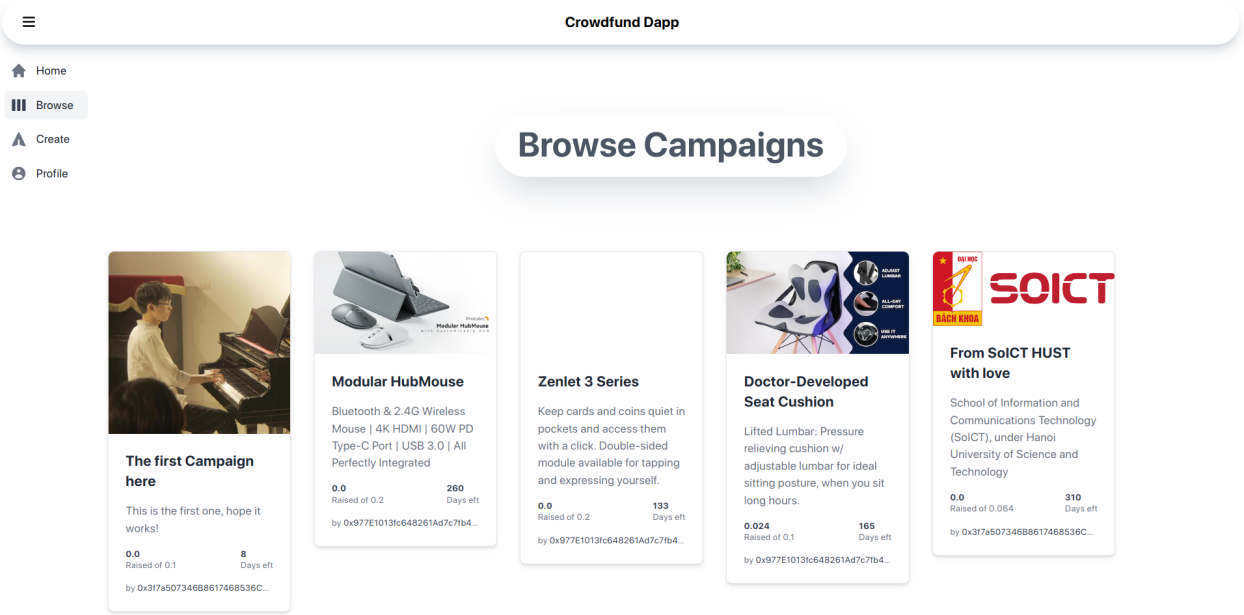
3.3. Home page



The Home page is the landing page of the app. It displays a call-to-action button that allows the user to connect their wallet. Once the wallet is connected, the button changes to display the user's wallet address.

The purpose of this page is to welcome the user and provide a way for them to connect their wallet, which is required to use certain features of the app.

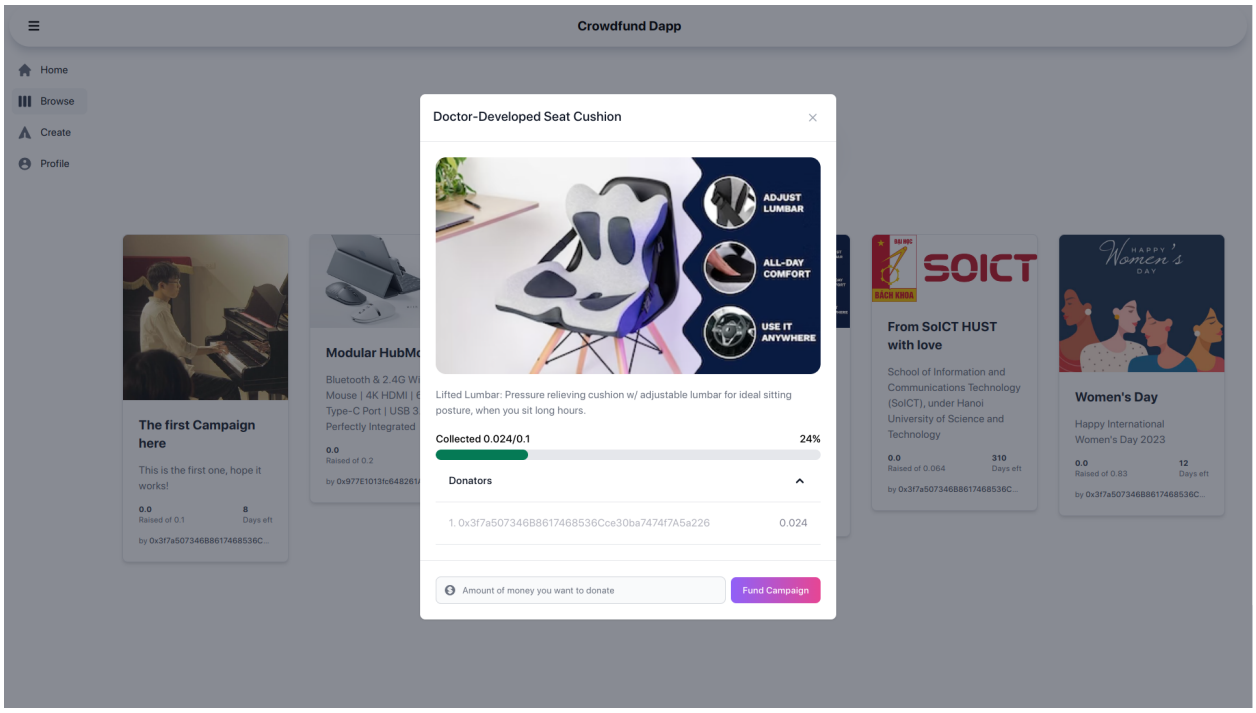
3.4. Browse page



The Browse page is where users can browse all available campaigns. This page is accessible even if the user has not connected their wallet. The page displays a list of campaigns, each with a title, image, and a brief description.

Get all campaign addresses from campaign storage
Create an empty list for campaign objects
FOR each campaign address
    Get the campaign contract instance using ether.js
    Get the campaign details from the contract instance
    Create a campaign object with the details
    Add campaign object to the list
Return the list of campaign objects

- Donate modal

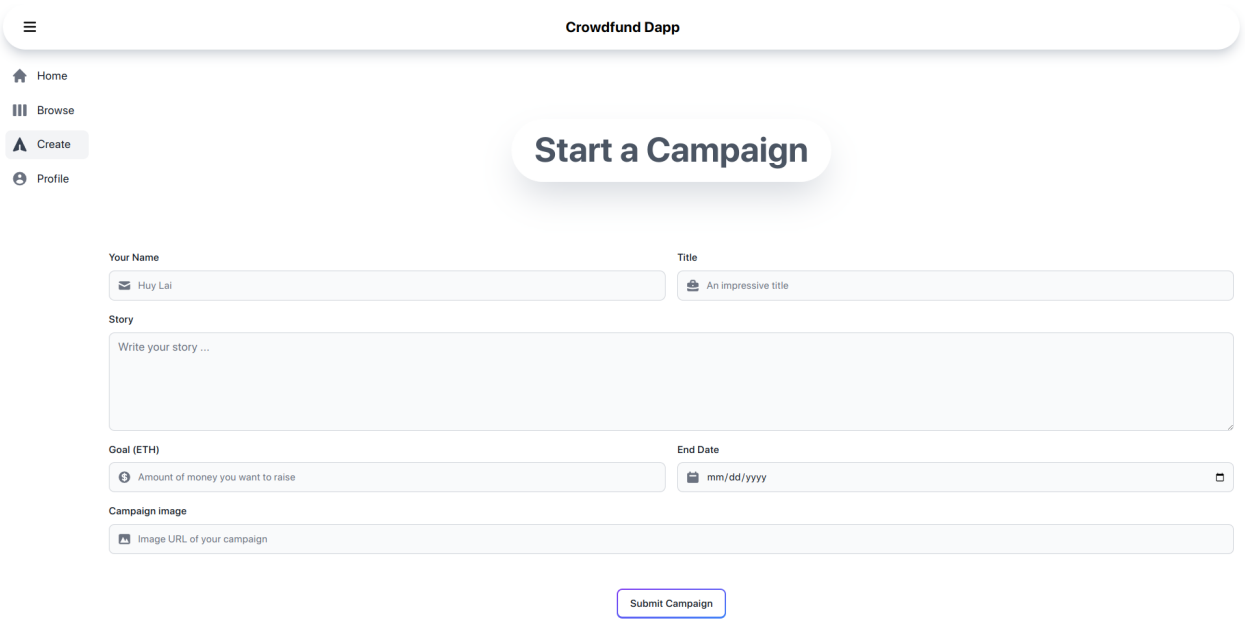


When the user clicks on a campaign, a modal appears that provides more details about the campaign. The modal displays the campaign's progress towards its funding goal, a form that allows the user to enter the amount they wish to donate, and a button that triggers the 'donate' function.

IF user is connected to a wallet
    Get the necessary data from the form inputs
    Click Submit Campaign button
    Call the createCampaign function from the smart contract, passing in the data as arguments
    Wait for the transaction to be confirmed
    Show a success message to the user

ELSE
    Show a failure message to the user

3.5. Create page

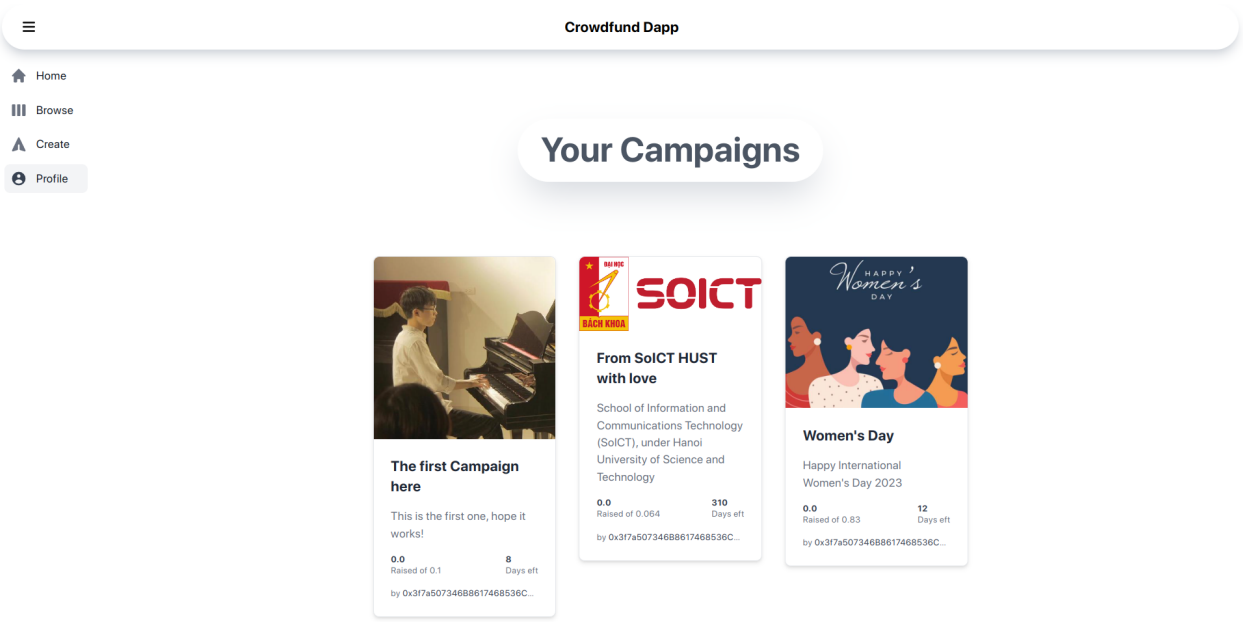


The Create page is where users can submit a new campaign. This page is accessible only if the user has connected their wallet. The page displays a form where the user can enter the title, description, target amount, deadline, and an image for their campaign.

When the user clicks the 'Submit Campaign' button, MetaMask prompts the user to confirm the transaction. If the user agrees, the app sends the information to the smart contract using the 'create' function, and a successful alert is displayed once the transaction is complete. If the user declines the transaction, a fail alert is displayed.

IF user is connected to a wallet
    Get the necessary data from the form inputs
    Click Submit Campaign button
    Call the createCampaign function from the smart contract, passing in the data as arguments
    Wait for the transaction to be confirmed
    Show a success message to the user
ELSE
    Show a failure message to the user

3.6. Profile page



The Profile page is where users can view all campaigns they have submitted. This page is accessible only if the user has connected their wallet. The page displays a list of campaigns owned by the user, each with a title, image, and a brief description.

The page functions similarly to the Browse page, but it filters only campaigns that have the same owner address as the connected account.

Get user's Ethereum address from MetaMask
Get all campaign addresses from campaign storage
Create an empty list for campaign objects
FOR each campaign address
    Get the campaign contract instance using ether.js
    If the campaign has the same owner address as the user
    Get the campaign details from the contract instance
    Create a campaign object with the details
    Add campaign object to the list
Return the list of campaign objects

VIII. Testing and Deployment

1. Testing the smart contract and the DApp

For the user interface and user experience design, it is important to test the navigation flow and the responsiveness of the pages on different devices. It is also important to test the user input validation and error handling. This can be done manually by testing each page and each input field to ensure that they work as expected.

For the functionality of the smart contract, it is important to test the create campaign and donate functions to ensure that they work as expected. This can be done using automated tests using a tool like Hardhat. The tests should cover different scenarios such as creating a campaign with different values for target and deadline, donating to a campaign with different amounts, and checking the balance of the campaign after donations have been made.

After testing is complete, the smart contract can be deployed to the Ethereum network using a tool like Truffle or Remix. It is important to ensure that the deployment is successful and that the smart contract is properly linked to the DApp. Finally, the DApp should be tested on the Ethereum test network to ensure that it works as expected, using a platform like ThirdWeb.

1.1. Hardhat

Hardhat is a popular development environment for building, testing, and deploying smart contracts on Ethereum. It was created in 2018 by the development team at Nomic Labs and is now maintained by a community of contributors.

Hardhat is known for its speed, flexibility, and extensibility. It allows developers to write tests for their smart contracts in Solidity or other languages, such as JavaScript or TypeScript, and run them in a local blockchain network. This helps to catch errors and bugs before deploying the contract to the main Ethereum network.

Hardhat is an open-source development environment for building and testing smart contracts on Ethereum. It was initially released in January 2020 and has gained popularity among Ethereum developers due to its ease of use, powerful features, and extensibility.

Hardhat is built on top of the Ethereum Virtual Machine (EVM) and provides a development environment that is specifically designed for Ethereum smart contract development. One of the key features of Hardhat is its ability to run and debug smart contracts locally, which allows developers to test their code before deploying it to the Ethereum network.

Hardhat also provides a number of other features that make it a powerful tool for Ethereum development. One of the key features of Hardhat is its ability to automate the process of deploying and testing smart contracts. Hardhat provides a built-in task runner that allows developers to write scripts to perform common tasks, such as compiling contracts, running tests, and deploying to a local or remote network.

Hardhat also includes a built-in console that allows developers to interact with their contracts in real-time. This is especially useful for debugging and testing smart contracts, as it allows developers to quickly see the results of their code changes.

Another feature of Hardhat is its support for multiple Ethereum networks, including the main Ethereum network, test networks, and private networks. This

allows developers to test and deploy their contracts in different environments and ensure that they work correctly under different conditions.

Another important feature of Hardhat is its support for automated testing, which allows developers to write tests that can be run automatically as part of the development process. Hardhat comes with its own testing framework, which is based on Mocha and Chai, and allows developers to write tests in JavaScript.

Hardhat is also highly extensible, with a plugin system that allows developers to add custom functionality to the environment. This has led to the creation of many community-developed plugins that add features such as code coverage, gas usage profiling, and more.

Overall, Hardhat has become a popular choice for Ethereum developers due to its speed, flexibility, and extensibility. It is widely used in the Ethereum ecosystem and has been adopted by many leading companies and projects.

1.2. Ethereum Test Network

Ethereum test networks are blockchain networks that are used for testing smart contracts and decentralized applications (dApps) before they are deployed on the Ethereum mainnet. These test networks are identical to the mainnet in terms of functionality and behavior, but the main difference is that they use test Ether instead of real Ether. This means that developers can test their dApps and smart contracts in a sandbox environment without risking any real money.

There are several Ethereum test networks available, each with its own purpose and features. The most commonly used test networks are:

- Ropsten: Ropsten is the oldest Ethereum test network and is widely used for testing dApps and smart contracts. It uses a proof-of-work consensus algorithm and has a similar block time and gas limit as the Ethereum mainnet.
- Rinkeby: Rinkeby is a proof-of-authority Ethereum test network that was designed to be more stable and faster than Ropsten. It is widely used for testing dApps and smart contracts that require fast confirmation times and low fees.
- Kovan: Kovan is a proof-of-authority Ethereum test network that is similar to Rinkeby in terms of stability and speed. It is used for testing dApps and smart contracts that require a more stable environment than Ropsten.
- Goerli: Goerli is a proof-of-authority Ethereum test network that was designed to be more decentralized and community-driven than other test networks. It is used for testing dApps and smart contracts that require a decentralized environment (we use this testnet in this project).

Developers can switch between different test networks using their Ethereum client or provider, such as MetaMask, Infura, or Alchemy. Testing on a test network is essential to ensure that smart contracts and dApps are functioning as intended before deploying them on the mainnet.

- More about Goerli Testnet

Goerli is a popular test network on the Ethereum blockchain. It's a proof-of-authority (PoA) network, which means that it uses a different consensus mechanism than the Ethereum mainnet, where miners compete to add blocks to the blockchain.

On the Goerli network, validators are chosen by the network's authorities. This makes it a more centralized network compared to the Ethereum mainnet. However, the use of PoA also makes the network faster and cheaper to use for testing and development purposes.

Goerli was launched in 2019 as a response to the need for a reliable test network that could support the testing and development of Ethereum-based applications. It's one of several test networks available, alongside Rinkeby, Kovan, and Ropsten.

To use the Goerli network, developers can connect to it using their Ethereum wallets or through APIs provided by services such as Infura or Alchemy. Once connected, developers can deploy and test their smart contracts and DApps in an environment that's similar to the Ethereum mainnet but without the high costs and slow transaction times.

In order to obtain Goerli test Ether (GöETH), developers can use a faucet service, which distributes test Ether for free to those who request it. This allows developers to test and deploy their applications without the need to spend real Ether on the mainnet.

Overall, Goerli is a useful and reliable test network that can help developers test and deploy their Ethereum-based applications with ease. Its fast transaction times and low costs make it an ideal choice for testing and development purposes.

1.2. Ethers.js

ethers.js is a popular JavaScript library for interacting with the Ethereum blockchain. It was first released in 2018 and is maintained by a team of developers from Nomic Labs.

The library provides a simple and intuitive interface for developers to interact with the Ethereum blockchain, including functionality for creating, signing, and sending transactions, interacting with smart contracts, and working with accounts and wallets. It also includes support for Ethereum Name Service (ENS), a decentralized domain name system on the Ethereum blockchain.

Ethers.js supports the latest Ethereum specifications and is designed to work seamlessly with the web3.js library, which was previously the most popular library for interacting with Ethereum. However, ethers.js has gained popularity due to its improved performance, security, and developer experience. It also includes TypeScript support, which provides additional safety and tooling for developers.

Some key features of ethers.js include:

1. Support for both Ethereum mainnet and test networks
2. Built-in support for common wallet formats, including MetaMask and Ledger
3. Easy integration with other Ethereum libraries and frameworks
4. Comprehensive documentation and an active community

Overall, ethers.js has become a popular choice for developers building decentralized applications on the Ethereum blockchain due to its robust features and ease of use.

2. Deployment of the DApp

2.1. ThirdWeb

ThirdWeb.com is a web-based platform that enables users to deploy and interact with decentralized applications (DApps) on the Ethereum network. The platform provides a simple and user-friendly interface for users to deploy their DApps without having to write any complicated code.

One of the main advantages of ThirdWeb.com is that it is a web-based platform, which means that users do not need to install any software or set up any infrastructure to deploy their DApps. The platform also provides users with a range of tools and services that they can use to test and debug their DApps before deploying them to the Ethereum network.

Another key feature of ThirdWeb.com is its support for smart contracts written in Solidity, which is a high-level programming language used to write smart contracts on the Ethereum network. The platform allows users to easily create, compile, and deploy Solidity smart contracts to the Ethereum network.

In addition to its deployment tools, ThirdWeb.com also provides users with a range of services to help them manage their DApps once they are deployed to the Ethereum network. These services include tools for monitoring and analyzing DApp performance, as well as tools for managing user accounts and transactions.

Overall, ThirdWeb.com is a powerful and user-friendly platform for deploying and managing DApps on the Ethereum network. Its web-based interface, support for Solidity smart contracts, and range of tools and services make it an ideal platform for developers and entrepreneurs looking to build and deploy DApps quickly and easily.

2.2. NextJS

Next.js is an open-source framework for building server-side rendered (SSR) React applications. It was created by Vercel (formerly Zeit) and released in 2016. Next.js combines the best of both worlds by providing developers with the flexibility of client-side rendering and the performance of server-side rendering.

One of the key features of Next.js is its built-in support for server-side rendering, which allows for faster page loads, better SEO, and improved performance. In addition, it provides developers with several other features that make building complex web applications easier, such as automatic code splitting, client-side routing, and dynamic imports.

Another important feature of Next.js is its support for static site generation (SSG), which allows for the generation of pre-rendered HTML pages at build time. This can greatly improve the performance and scalability of web applications, especially for content-heavy sites.

Next.js also comes with a number of pre-built features and optimizations, including serverless functions, optimized images, and automatic resource optimization, which help to streamline the development process and improve performance.

One of the biggest advantages of Next.js is its strong community support and the availability of numerous plugins, extensions, and packages that can be easily integrated into your application.

In summary, Next.js is a powerful and flexible framework for building fast and scalable web applications, with built-in support for server-side rendering, static site generation, and a range of other features and optimizations.

IX. Evaluation and Conclusion

1. Evaluation of the decentralized crowdfunding app

Evaluation of the decentralized crowdfunding app:
The decentralized crowdfunding app developed in this project provides a platform for users to create and fund campaigns using Ethereum smart contracts. The app uses a user-friendly interface built with NextJS and connects to the Ethereum network using Ether.js. The app provides a secure and transparent way for users to create and fund campaigns without the need for a centralized intermediary.

The smart contract code used in the app was tested extensively using Hardhat, and the app was deployed on the Goerli test network to ensure that it operates smoothly and securely. The app has several features, such as the ability to browse campaigns, create new campaigns, and view user profiles, that are essential for a crowdfunding platform.

However, there are areas where the app could be improved. For example, the app currently only supports the Ethereum network, and it could benefit from adding support for other blockchain networks. Additionally, the app could be enhanced to provide more analytics and insights into the performance of campaigns, as well as provide a more seamless user experience for the donation process.

2. Conclusion and future work

In conclusion, this project demonstrated how decentralized crowdfunding can be implemented using Ethereum smart contracts and a user-friendly interface built with NextJS. The app provides a secure and transparent way for users to create and fund campaigns without the need for a centralized intermediary.

Future work could include expanding the app to support other blockchain networks, adding more analytics and insights into campaign performance, and improving the user experience for the donation process. Additionally, the app could be extended to include features such as automatic refunds in the case of campaign failures, multi-currency support, and the ability to set up recurring donations. With these enhancements, the app could become a more comprehensive and versatile platform for decentralized crowdfunding.

X. References

1. "Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto (2008)
2. "Mastering Blockchain: Distributed ledger technology, decentralization, and smart contracts explained" by Imran Bashir (2018)
3. "Ethereum: A Secure Decentralized Generalized Transaction Ledger" by Vitalik Buterin (2014)
4. "Blockchain Basics: A Non-Technical Introduction in 25 Steps" by Daniel Drescher (2017)
5. "Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World" by Don Tapscott and Alex Tapscott (2016)
6. "Mastering Blockchain: Distributed ledger technology, decentralization, and smart contracts explained" by Imran Bashir (2018)
7. "Smart Contracts: The Essential Guide to Using Blockchain Smart Contracts for Cryptocurrency Exchange, Smart Contracts Investing, and Security Tokens" by David J. Brodsky (2018)
8. "Mastering Ethereum: Building Smart Contracts and DApps" by Andreas M. Antonopoulos and Gavin Wood (2018)
9. "Solidity Programming Essentials: A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain" by Ritesh Modi (2018)
10. Solidity documentation: https://docs.soliditylang.org/
11. Ethereum whitepaper: https://ethereum.org/en/whitepaper/
12. Ethereum Test Networks: https://eth.wiki/en/networks/testnets
13. Goerli Testnet: https://goerli.net/
14. Ethers.js official website: https://docs.ethers.io/v5/
15. Hardhat documentation: https://hardhat.org/getting-started/
16. ThirdWeb documentation: https://docs.thirdweb.com/

XI. Application Demo

1. Live demo

The app has been deployed at laiquanghuy-dapp.vercel.app
Anyone can go to this website to try to create, browse and donate to campaigns.
Remember to connect to your wallet first.

2. Local environment setup

Go to the frontend directory in the terminal, make sure pnpm is installed.

```
cd frontend
pnpm i
pnpm dev
```

Go to localhost:3000 on your browser to use the app.