APRIL 28, 2021

Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Decentralised Application for Crowdfunding on a Blockchain

TECHNICAL GUIDE

Student: Karl Whelan
Supervisor: Geoff Hamilton
Student Number: 15561423

## Contents

1. Introduction ..... 3
   1.1. Abstract ..... 3
   1.2. Motivation ..... 3
   1.3. Project Context ..... 4
   1.4. Research ..... 4
   1.5. Glossary ..... 5
2. Design ..... 7
   2.1. General Design ..... 7
      2.1.1. User Interface ..... 7
      2.1.2. Backend ..... 11
   2.2. System Architecture ..... 12
   2.3. High level Design ..... 13
      2.3.1. Context diagram ..... 13
      2.3.2. Use case sequence diagrams. ..... 14
3. Implementation ..... 16
   3.1. Smart Contract ..... 16
      3.1.1. Structures ..... 16
      3.1.2. Create Project ..... 18
      3.1.3. Return list of created projects to the user interface ..... 19
      3.1.4. Donate to a project ..... 19
      3.1.5. Check if a project has ended or the funding goal has been reached ..... 20
      3.1.6. Pay-out to project creator ..... 20
      3.1.7. Pay refund to all projects donators ..... 21
      3.1.8. Return contract balance ..... 21
   3.2. User Interface Functions ..... 22
      3.2.1. Connect to web3 and IPFS ..... 22
      3.2.2. Retrieve project list from blockchain ..... 23
      3.2.3. Create a project ..... 24
      3.2.4. Donate to a project ..... 25
      3.2.5. Miscellaneous ..... 26
   3.3. User Interface Components ..... 27
      3.3.1. Header and Navbar ..... 28
      3.3.2. Homepage ..... 30
      3.3.3. Create Project Page ..... 31
      3.3.4. View Projects Page ..... 33

3.3.5. View Project ... 34
3.3.6. Input ... 35
4. Testing ... 36
4.1. Smart contract unit and end-to-end testing ... 36
4.1.1. Testing contract deployment ... 37
4.1.2. Testing contract creation ... 38
4.1.3. Testing return project's function ... 39
4.1.4. Testing donate to project function ... 40
4.1.5. Testing the advance time helper function ... 41
4.1.6. Testing mock project ending by date ... 42
4.1.7. End-to-end pay out to project creator test ... 43
4.1.8. End-to-end pay refunds test ... 44
4.2. Smart contract scalability and gas usage testing ... 45
4.2.1. Building the maps for our log files ... 46
4.2.2. Creating projects function ... 47
4.2.3. Donate to single project multiple times ... 48
4.2.4. Donate to multiple projects ... 49
4.3. User interface testing ... 51
4.4. Usability testing and user feedback ... 54
5. Conclusion ... 57
5.1. Problems solved ... 57
5.2. Future work ... 57

# 1. Introduction

## 1.1. Abstract

Crowdfunding on the web is very popular these days. There are trust issues with crowdfunding currently though. Most crowdfunding projects do not deliver on time and some never deliver at all. Crowdfunding platforms also act as middlemen you must trust to manage the transfer of funds appropriately. These platforms charge fees for this service, generally between 3% - 5% of all raised funding.

Blockchain technology offers good solutions to these problems. On a blockchain every transaction is traceable and transparent which helps with trust. Smart contracts with predefined rules can dictate how money is held and transferred removing the need to trust middlemen or pay their fees. The decentralised and encrypted nature of blockchains also protects against malicious actors and single points of failure. This is doubly important when managing sums of money.

This project is a fully decentralised crowdfunding platform built using blockchain technology and smart contracts. All transfers of funds are handled by smart contracts deployed on a blockchain while project information is stored to the InterPlanetary File System, (IPFS. A peer-to-peer decentralised file system), to save on gas without sacrificing decentralisation or security. Includes a React web app user interface for users to create, donate to and display projects.

This decentralised application, (Dapp), can be run on the Ethereum Mainnet, any of its test networks or a personal ganache blockchain.

A live version of the UI connected to the Rinkeby test network is available here: https://main.d3oz5l3o8ahlw9.amplifyapp.com/

## 1.2. Motivation

My primary ambition with this project was to learn about blockchain technology, smart contracts, and decentralised applications. Ethereum launched in 2015 allowing the execution of smart contracts on the Ethereum Virtual Machine (EVM). This in turn allowed users to create fully decentralised applications which benefited from the security, traceability, and transparency of blockchains. Since then, decentralized application development has become more and more popular with more novel applications being developed. Ethereum 2.0 is set to launch sometime in the near future and promises to implement a proof-of-stake method, to validate and record transactions, and introduce data sharding. These changes will drastically improve the efficiency, speed, and scalability of Ethereum further increasing its usability and increase the number of use cases for applications it will be suited for. These upcoming improvements, the inevitable future improvements to the technology overtime, and the rising popularity of cryptocurrencies in general mean blockchains and decentralised applications are likely to be increasingly relevant in the future.

I chose a crowdfunding platform for my project as I felt there were a lot of issues with crowdfunding that blockchain technology could help with as I mentioned in the abstract above. Another ambition with the project was to learn more about user interface development and I felt a crowdfunding platform was a project that could include a user facing web app with a clear scope of the functionality it would need.

## 1.3. Project Context

The application is intended to be used by anyone who is interested in getting personal projects financed or finding projects they are interested in backing. An understanding of, or an interest in, blockchain technology is not required to use the application. The user interface should abstract enough complexity from the execution of the smart contracts while still giving the user enough feedback so that it is clear to them what is happening.

## 1.4. Research

This project used a continuous development approach meaning there was a large amount of informal research carried out weekly as new issues were identified, and new features were implemented. As such the list of resources is too long to list here but I will list some major ones that were used often.

- Ethereum Yellow Paper: A formal specification of the Ethereum protocol and the Ethereum state machine.

- Ethereum developer documentation: Provides documentation for Dapp developers explaining concepts such as web3, block mining, gas usage, the Ethereum tech stack and much more.

- Solidity docs: Documentation for the solidity programming language. This is the language I used to write the smart contract.

- truffle docs: Documentation for the truffle developer environment which I used to compile, deploy, and test the smart contract.

- React docs: Documentation for React, the JavaScript library I used to make the user interface.

- React Tutorial: A beginner bootcamp style tutorial from free code camp explaining the concepts underpinning React.

- React Testing Tutorial: A tutorial about the concepts around testing React components.

- InterPlanetary File System docs: Docs explaining what IPFS is and how it works.

- CSS docs: Documentation for CSS styling.

## 1.5. Glossary

- Blockchain: A decentralised database consisting of blocks that are linked using cryptography. Secure by design. Typically used as a distributed ledger, managed by a peer-to-peer network adhering to a protocol for block validation and node communication.

- Smart contracts: A self-executing computer program which implements the terms of an agreement between 2 parties.

- Decentralised Application: An application that runs on a distributed computer system. In this case the Ethereum Virtual Machine.

- Ethereum: An open source blockchain that features smart contract support. Also, the name of the protocol(s) used for block validation and node communication on this blockchain.

- Ethereum Virtual Machine: A software platform running on the Ethereum blockchain. It is a distributed state machine.

- Eth: Short for Ether. This is the cryptocurrency built on top of Ethereum. It acts as “fuel” allowing smart contracts to run.

- Geth: Short for Go Ethereum. It is a client that implements the Ethereum protocol allowing a machine to become a node on the Ethereum network, validate blocks and interact with other nodes.

- Web3.js: A JavaScript API that allows you to interact with a local or remote Ethereum node using HTTP.

- Solidity: A programming language for writing smart contracts.

- Truffle: A development environment, testing framework and asset pipeline for blockchain development.

- Ganache: A personal blockchain that can be run locally to test smart contract functionality and deployment.

- Rinkeby: A public test blockchain where are currency used for transactions is fake.

- Gas: Refers to a fee paid to miners as an incentive for them to mine blocks. Mining blocks completes transactions and executes contracts.

- IPFS: The InterPlanetary File System. A protocol and peer-to-peer network for storing data in a decentralised file system. This allows us to store data in a way that does not cost gas or sacrifice the decentralisation that blockchains afford. All data stored here is immutable, so it does not sacrifice the security of blockchains either.

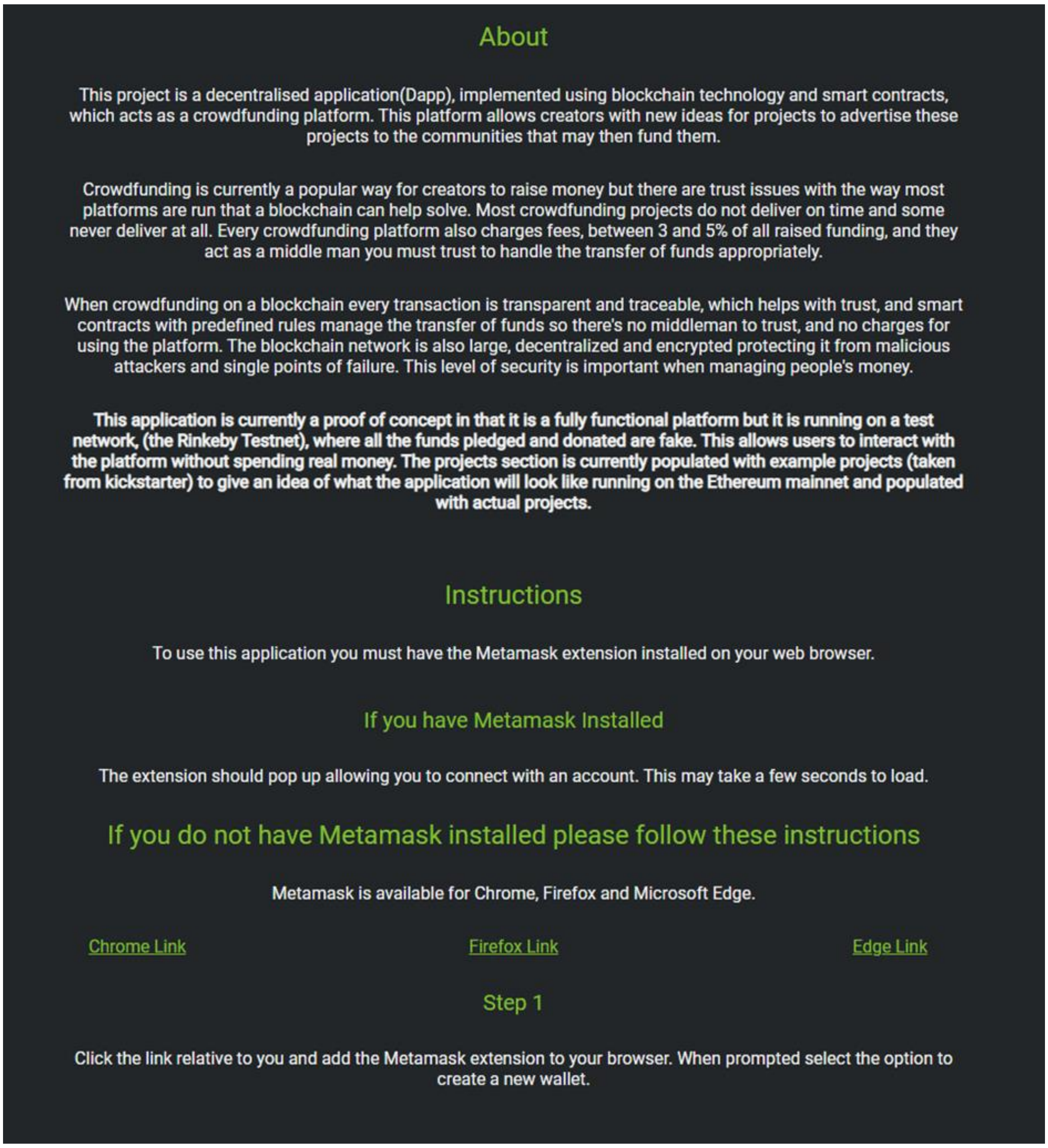
# 2. Design

## 2.1. General Design

### 2.1.1. User Interface

For the UI I decided to use React which is a JavaScript library for building UI components. React allows the building of responsive UIs that render modular components based on state. I thought this would benefit the responsiveness of the UI while also giving me an opportunity to learn a new system of building UIs. Web3 is the JavaScript library that I use to communicate with blockchain nodes, and it works well in a React app. React also let me develop an app that is suitable for multiple platforms.

I wanted the UI to be easy to understand and easy to navigate for all users. A user does not necessarily need an understanding of, or an interest in, blockchain technology to use this app. It should be usable by anyone with an interest in getting personal projects funded or finding interesting projects to donate to. Users need to install the Metamask extension, this allows them to connect to a blockchain without having to download a personal node for that blockchain. If a user does not have Metamask, simple step by step instructions on the landing page will help them install and use it.



## About

This project is a decentralised application(Dapp), implemented using blockchain technology and smart contracts, which acts as a crowdfunding platform. This platform allows creators with new ideas for projects to advertise these projects to the communities that may then fund them.

Crowdfunding is currently a popular way for creators to raise money but there are trust issues with the way most platforms are run that a blockchain can help solve. Most crowdfunding projects do not deliver on time and some never deliver at all. Every crowdfunding platform also charges fees, between 3 and 5% of all raised funding, and they act as a middle man you must trust to handle the transfer of funds appropriately.

When crowdfunding on a blockchain every transaction is transparent and traceable, which helps with trust, and smart contracts with predefined rules manage the transfer of funds so there's no middleman to trust, and no charges for using the platform. The blockchain network is also large, decentralized and encrypted protecting it from malicious attackers and single points of failure. This level of security is important when managing people's money.

**This application is currently a proof of concept in that it is a fully functional platform but it is running on a test network, (the Rinkeby Testnet), where all the funds pledged and donated are fake. This allows users to interact with the platform without spending real money. The projects section is currently populated with example projects (taken from kickstarter) to give an idea of what the application will look like running on the Ethereum mainnet and populated with actual projects.**

## Instructions

To use this application you must have the Metamask extension installed on your web browser.

### If you have Metamask Installed

The extension should pop up allowing you to connect with an account. This may take a few seconds to load.

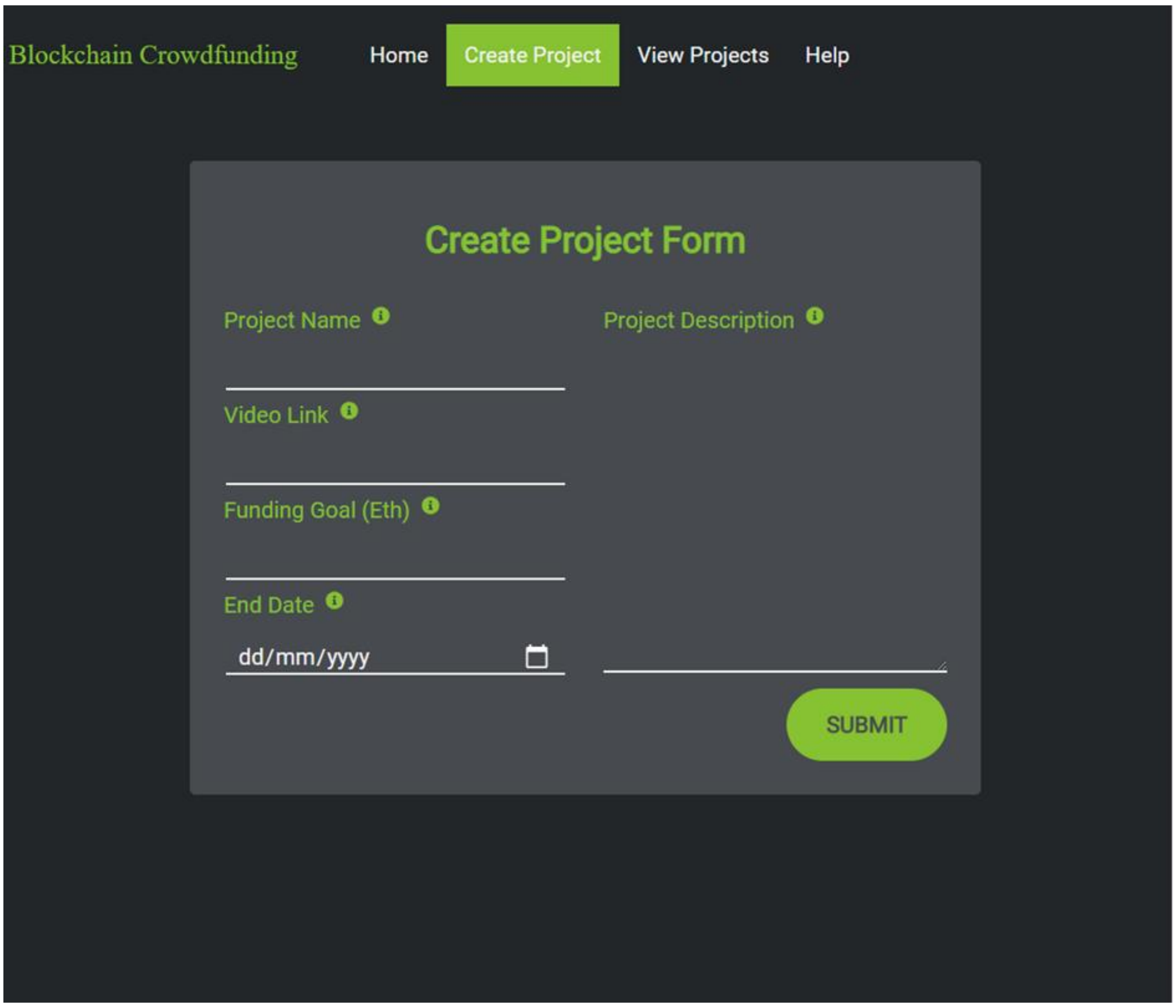
### If you do not have Metamask installed please follow these instructions

Metamask is available for Chrome, Firefox and Microsoft Edge.

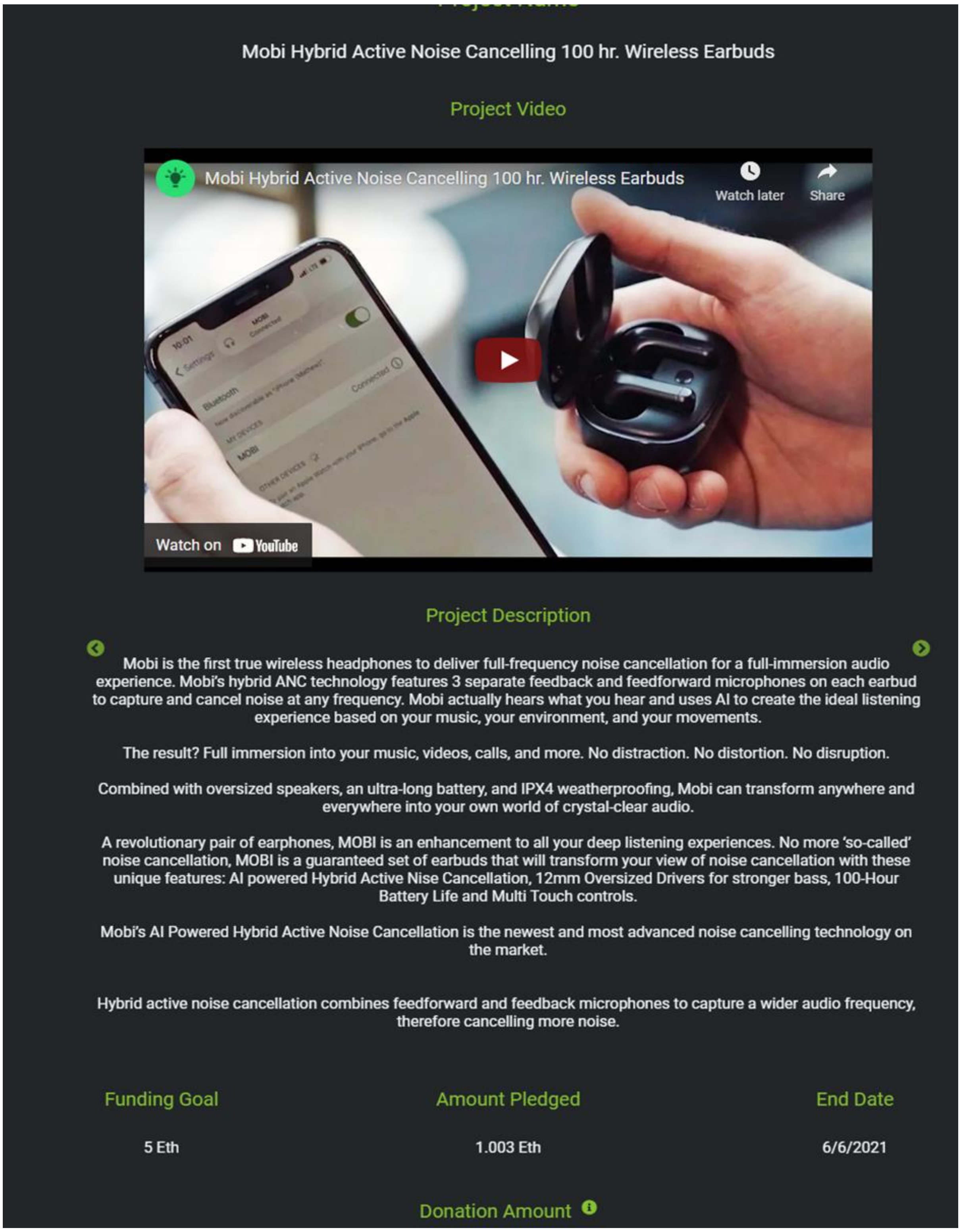
### Step 1

Click the link relative to you and add the Metamask extension to your browser. When prompted select the option to create a new wallet.

Once connected through Metamask the user can navigate to various views or pages using the responsive navbar. The create project page contains a simple form for creating a project. Tool tip icons give the user information about each field of the form. After submitting the form, they are asked for confirmation in Metamask. After they confirm the transaction, the block needs to be mined for the transaction to be complete. The user will be given feedback that the block is mining and weather the transaction was successful or not.



The view projects page contains a scrollable view of all active projects on the platform. Here users can see the projects name, video, description, funding goal, amount raised and end date. The user can scroll through projects using clickable left, right arrows or by dragging the screen across with the mouse or finger if on mobile. There is a donation field and button at the bottom of each individual project view allowing users to give a donation to that project. Like project creation they will be asked to confirm through Metamask and given feedback about block mining and if the transaction was successful or not.


Mobi Hybrid Active Noise Cancelling 100 hr. Wireless Earbuds
Watch later
Share
Watch on YouTube

Mobi Hybrid Active Noise Cancelling 100 hr. Wireless Earbuds

Project Video

Project Description

Mobi is the first true wireless headphones to deliver full-frequency noise cancellation for a full-immersion audio experience. Mobi's hybrid ANC technology features 3 separate feedback and feedforward microphones on each earbud to capture and cancel noise at any frequency. Mobi actually hears what you hear and uses AI to create the ideal listening experience based on your music, your environment, and your movements.

The result? Full immersion into your music, videos, calls, and more. No distraction. No distortion. No disruption.

Combined with oversized speakers, an ultra-long battery, and IPX4 weatherproofing, Mobi can transform anywhere and everywhere into your own world of crystal-clear audio.

A revolutionary pair of earphones, MOBI is an enhancement to all your deep listening experiences. No more 'so-called' noise cancellation, MOBI is a guaranteed set of earbuds that will transform your view of noise cancellation with these unique features: AI powered Hybrid Active Nise Cancellation, 12mm Oversized Drivers for stronger bass, 100-Hour Battery Life and Multi Touch controls.

Mobi's AI Powered Hybrid Active Noise Cancellation is the newest and most advanced noise cancelling technology on the market.

Hybrid active noise cancellation combines feedforward and feedback microphones to capture a wider audio frequency, therefore cancelling more noise.

| Funding Goal | Amount Pledged | End Date |
|---|---|---|
| 5 Eth | 1.003 Eth | 6/6/2021 |

Donation Amount

There is also a help page with some more information about the application including how to get test ether to use on the app.

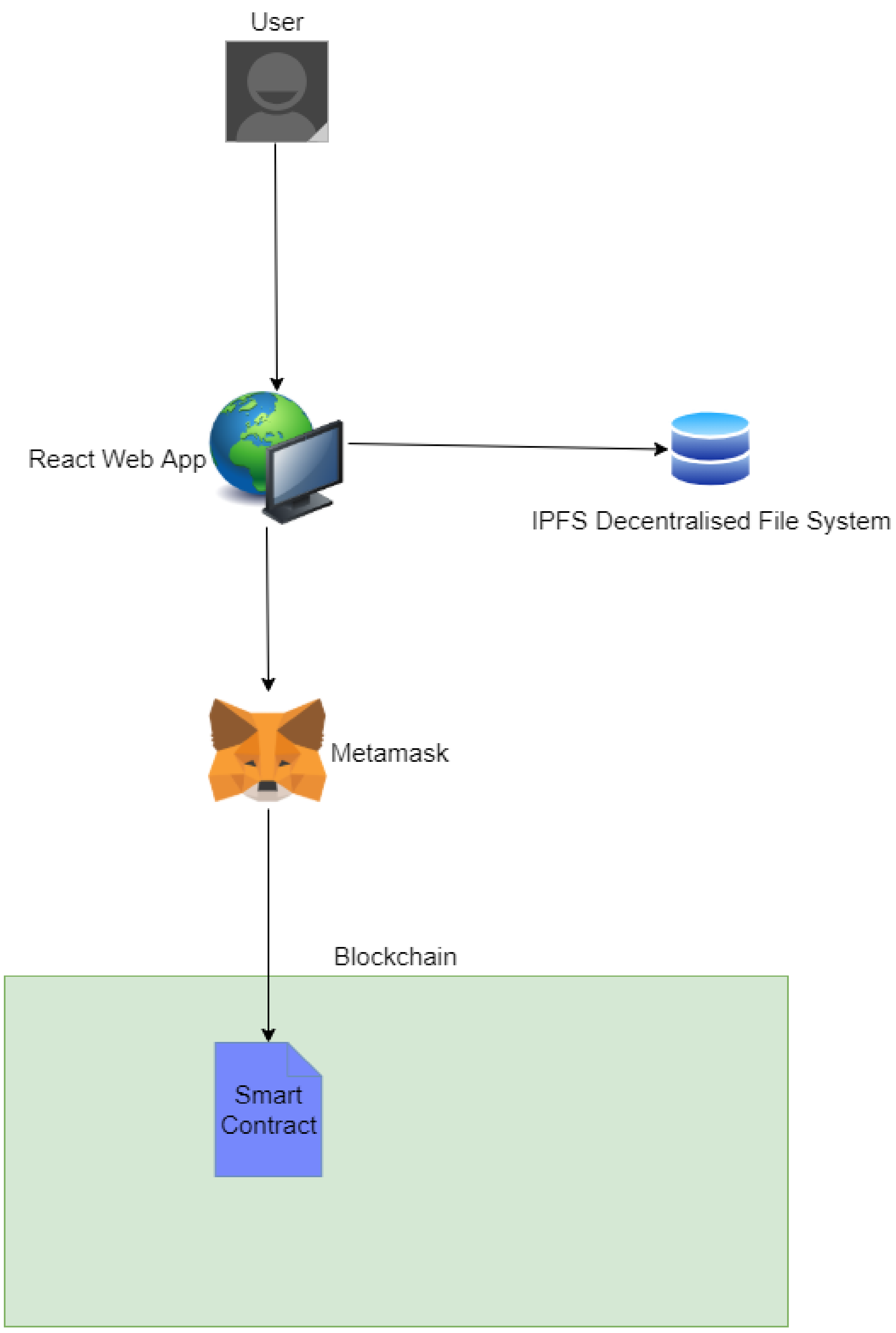
### 2.1.2. Backend

All the backend functionality is handled by a smart contract written in solidity. This contract manages the creation of projects, and donation to projects. It manages the paying out of funds either to the project creator when the funding goal has been reached, or refunding all the donators if the end date is reached. It also has view only functions, which do not cost gas, that return information to the frontend for display.

Storing all the project information on the blockchain would be prohibitively expensive. To avoid this, we store all the information that the blockchain does not need to know to execute its functions, (that is the project name, project description and the video link to the project video), to IPFS. IPFS stands for the InterPlanetary File System. It is a peer-to-peer network and protocol that allows saving data on a decentralised file system. For the purposes of this application, it allows me to save data without costing gas. Due to the decentralised and immutable nature of IPFS we can save this data without sacrificing any of the security and decentralisation that the blockchain affords. When saving data to IPFS you receive a hash which you can use to call your data like a key value pair. IPFS is very fast so the app can do this periodically without adding much latency. So instead of storing all that data on the blockchain we store it on IPFS and store the hash on the blockchain instead.

For the live app that I will be demoing the smart contract is compiled and deployed on the Rinkeby testnet. It could also be deployed on the Ethereum Mainnet, any of its testnet or on a personal blockchain if the user prefers.

## 2.2. System Architecture

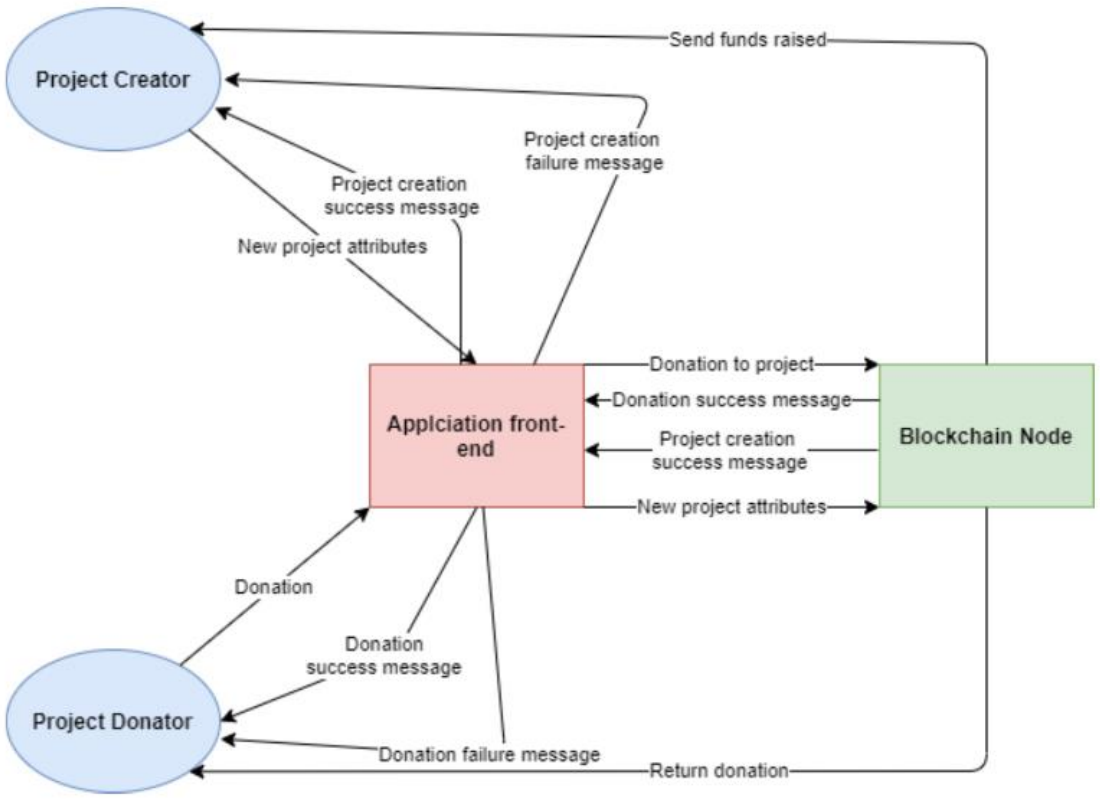


Here you see the system architecture diagram for the decentralised application. The user interacts with the React web app that abstracts the complexity of the system from them.

Data that is not required for the functions in the smart contract are stored on IPFS. Metamask allows connection to the smart contracts address on the blockchain and enables the sending of web3 messages.

## 2.3. High level Design

### 2.3.1. Context diagram

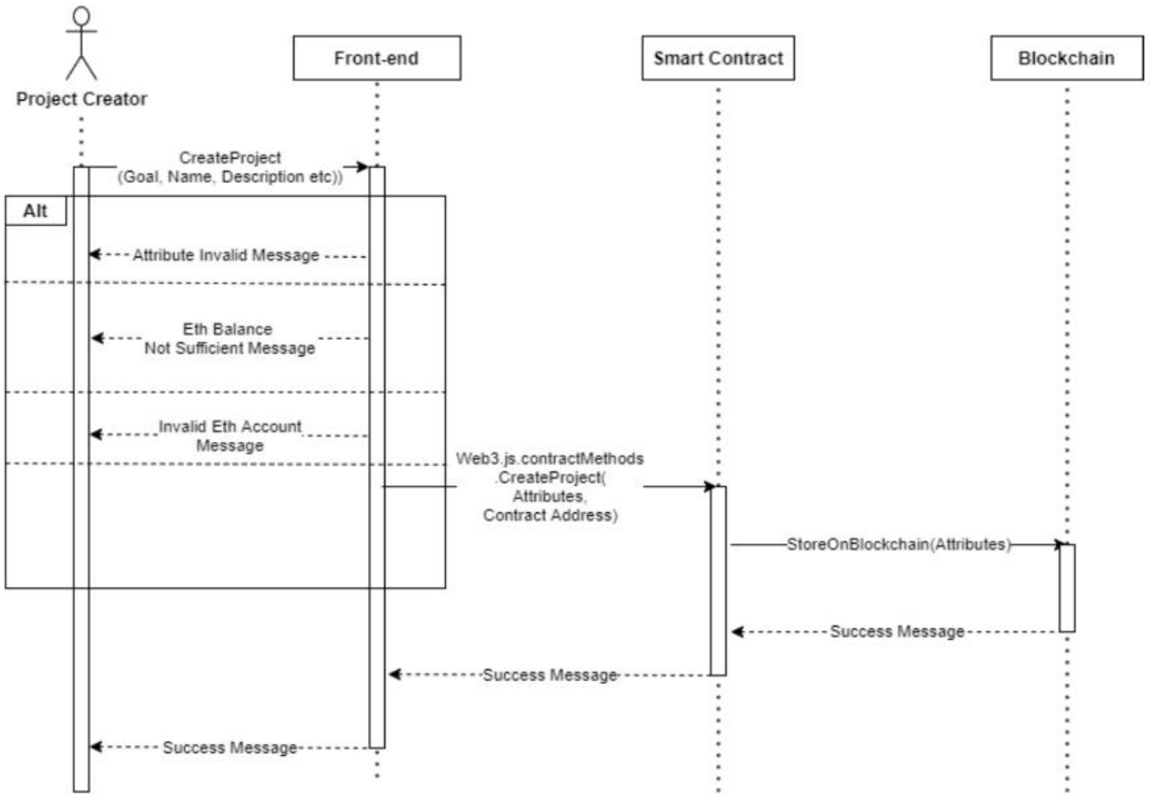


This context diagram shows the high-level design of the system.

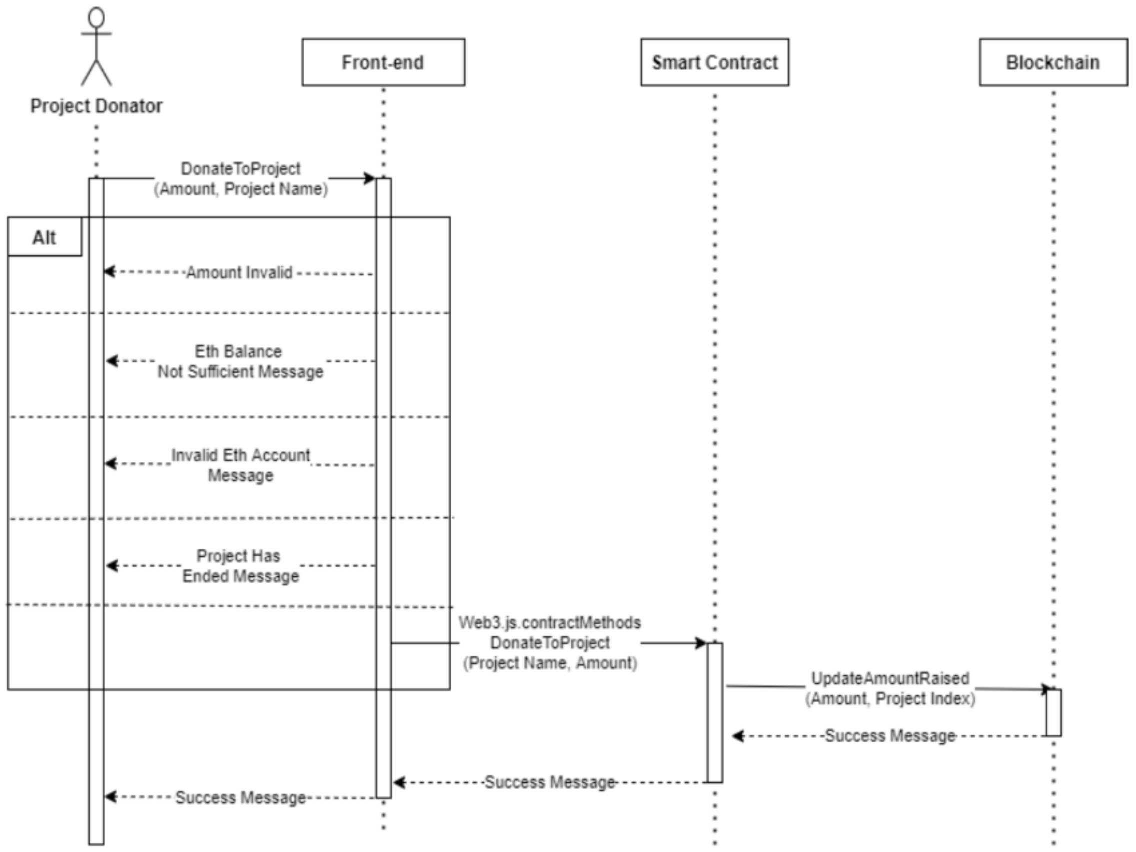
## 2.3.2. Use case sequence diagrams.

Below are 2 sequence diagrams showing the interaction between entities for both the create project and donate to project use cases.

### *2.3.2.1. Create project*


Project Creator
Front-end
Smart Contract
Blockchain
CreateProject (Goal, Name, Description etc))
Alt
Attribute Invalid Message
Eth Balance Not Sufficient Message
Invalid Eth Account Message
Web3.js.contractMethods .CreateProject( Attributes, Contract Address)
StoreOnBlockchain(Attributes)
Success Message
Success Message
Success Message

#### *2.3.2.2. Donate to project*



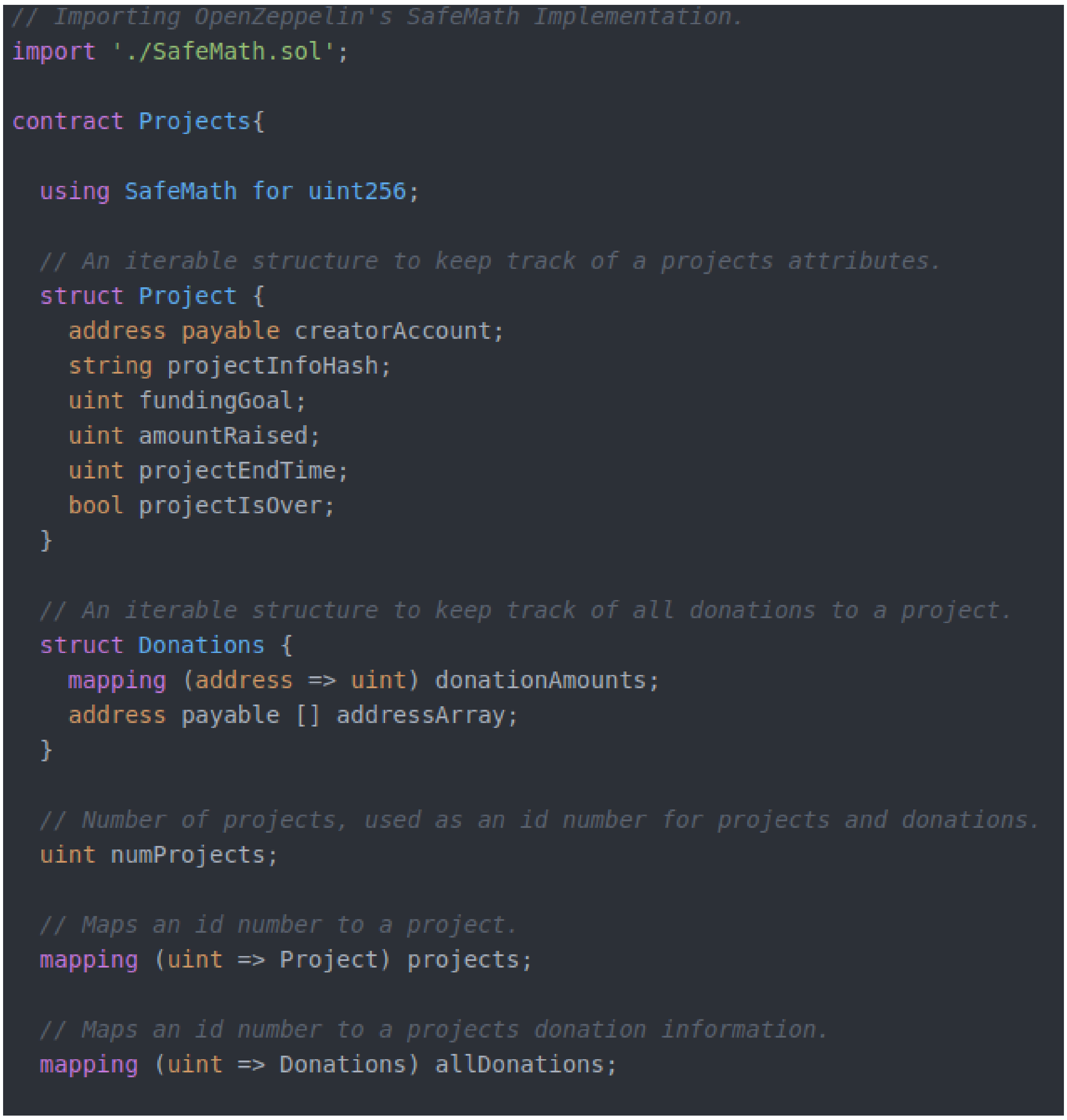
# 3. Implementation

This section contains a full explanation of the implementation of the project including sample code. I have broken it up into 3 sections.

## 3.1. Smart Contract

This section explains the layout and functions of the smart contract.

### 3.1.1. Structures



```
// Importing OpenZeppelin's SafeMath Implementation.
import './SafeMath.sol';

contract Projects{

  using SafeMath for uint256;

  // An iterable structure to keep track of a projects attributes.
  struct Project {
    address payable creatorAccount;
    string projectInfoHash;
    uint fundingGoal;
    uint amountRaised;
    uint projectEndTime;
    bool projectIsOver;
  }

  // An iterable structure to keep track of all donations to a project.
  struct Donations {
    mapping (address => uint) donationAmounts;
    address payable [] addressArray;
  }

  // Number of projects, used as an id number for projects and donations.
  uint numProjects;

  // Maps an id number to a project.
  mapping (uint => Project) projects;

  // Maps an id number to a projects donation information.
  mapping (uint => Donations) allDonations;
```

Here is the start of the Projects.sol smart contract containing the structures and variables that are used in the functions.

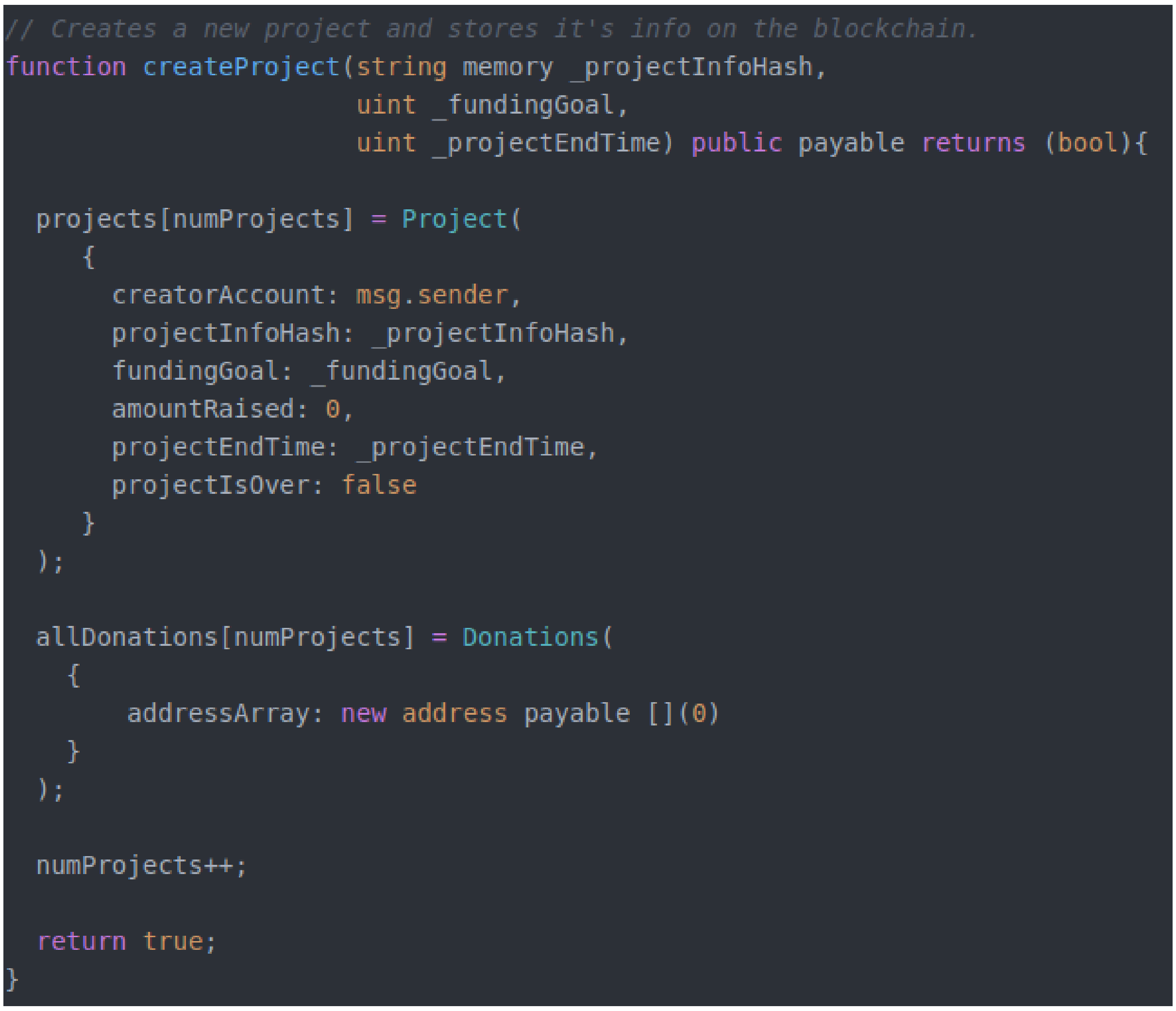
As you can see, I import SafeMath.sol and use it for unsigned integers. This is third party code that prevents overflows when doing arithmetic with large integers. This is a feature of most higher-level programming languages but not solidity so this code just ads that feature.

I have a structure called Project. This contains all the information about a project that the frontend needs to display. It contains the account address of the project creator, the hash value of the info stored on IPFS, the funding goal of the project in Eth, the amount raised in Eth, the end date of the project as a Unix timestamp and a Boolean that defaults as false that states weather the project has ended. The creator address must be saved as payable so that the pay-out function will work if it is called.

The Donations structure contains an array of the account addresses of all donators to a single project and a mapping from their address to the amount they have donated. Used in the case of paying refunds.

numProjects is just the number of created projects on the platform. This number is also used, on project creation, as an id number for Projects and Donations. The projects and allDonations mappings, map an id number to a Project and a Donations structure.

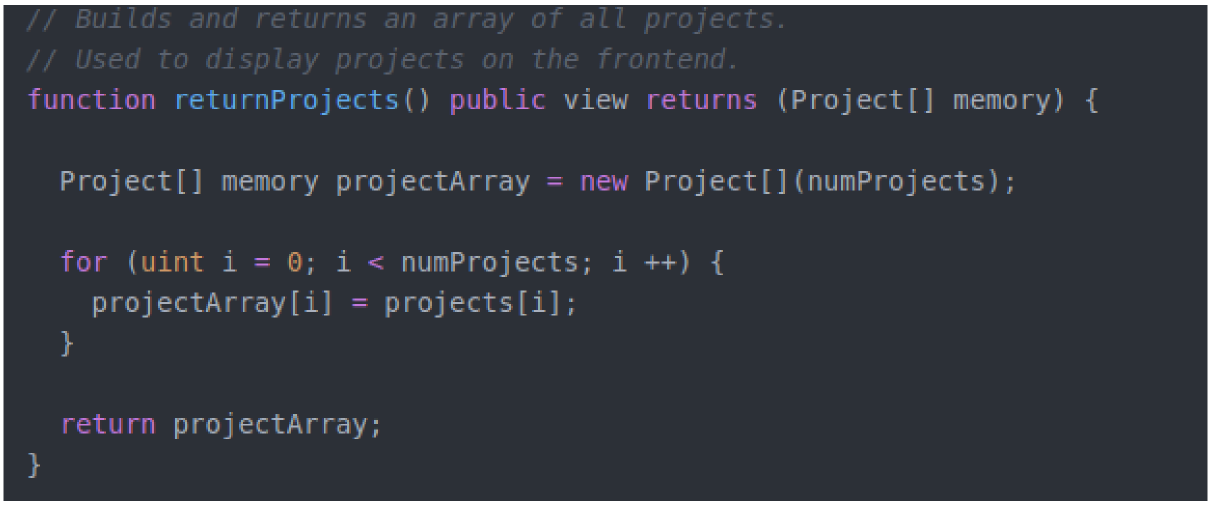
### 3.1.2. Create Project



```
// Creates a new project and stores it's info on the blockchain.
function createProject(string memory _projectInfoHash,
                       uint _fundingGoal,
                       uint _projectEndTime) public payable returns (bool){

  projects[numProjects] = Project(
     {
       creatorAccount: msg.sender,
       projectInfoHash: _projectInfoHash,
       fundingGoal: _fundingGoal,
       amountRaised: 0,
       projectEndTime: _projectEndTime,
       projectIsOver: false
     }
  );

  allDonations[numProjects] = Donations(
    {
        addressArray: new address payable [](0)
    }
  );

  numProjects++;

  return true;
}
```

This function takes a IPFS hash, an Eth funding goal, and a date in the form of a Unix timestamp and adds a project to the blockchain. It uses numProject as an ID to map to a Project structure and Donations structure. The project creator is the message sender to that is added as the creator account. A new empty address array is created for future donations.

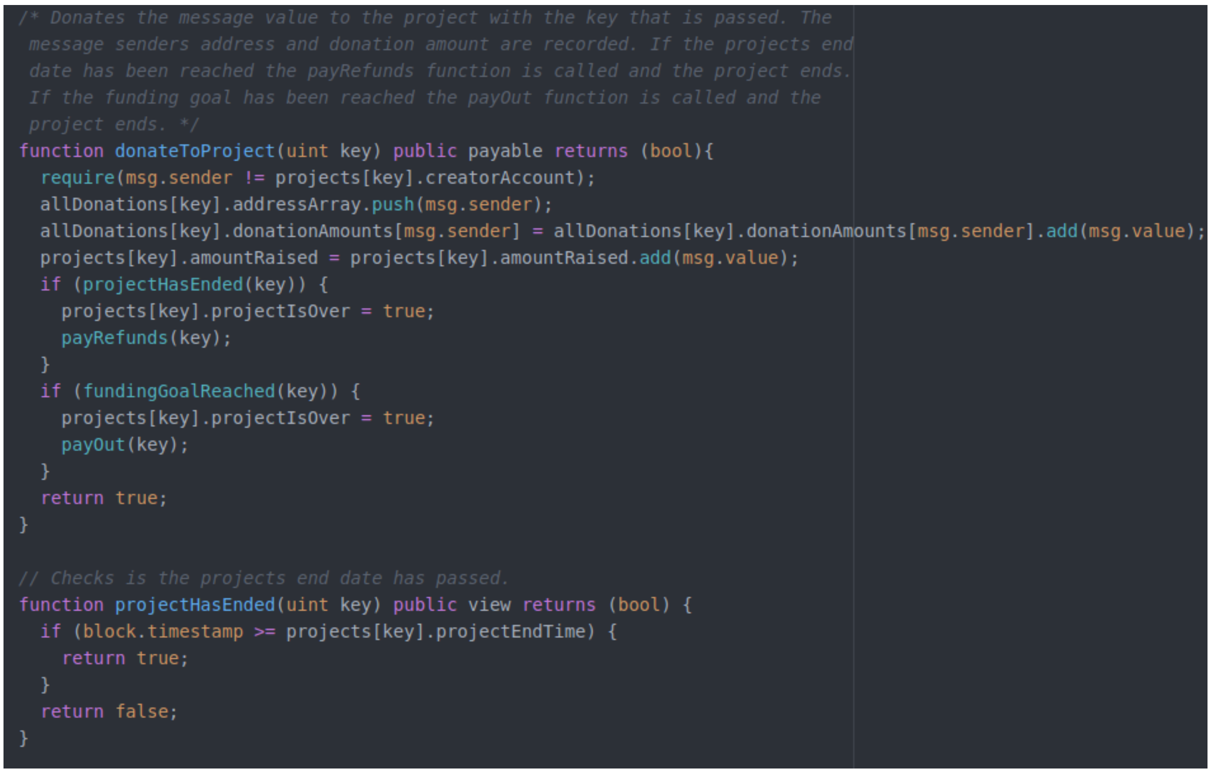
### 3.1.3. Return list of created projects to the user interface



```
// Builds and returns an array of all projects.
// Used to display projects on the frontend.
function returnProjects() public view returns (Project[] memory) {

  Project[] memory projectArray = new Project[](numProjects);

  for (uint i = 0; i < numProjects; i ++) {
    projectArray[i] = projects[i];
  }

  return projectArray;
}
```

This function is called by the frontend in order to fetch all created projects. It builds an array, adds each Project structure to that array and then returns the array. It is a view only function as it does not change or add anything to the blockchain. For this reason, it does not cost gas to use.

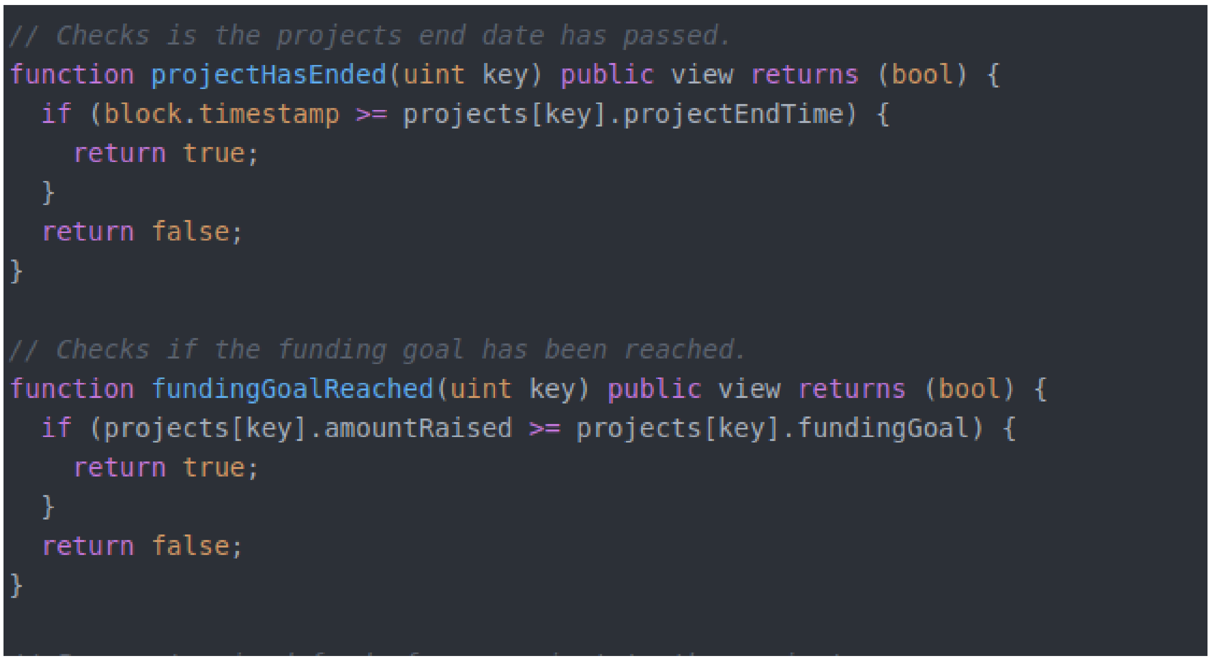
### 3.1.4. Donate to a project



```
/* Donates the message value to the project with the key that is passed. The
 message senders address and donation amount are recorded. If the projects end
 date has been reached the payRefunds function is called and the project ends.
 If the funding goal has been reached the payOut function is called and the
 project ends. */
function donateToProject(uint key) public payable returns (bool){
  require(msg.sender != projects[key].creatorAccount);
  allDonations[key].addressArray.push(msg.sender);
  allDonations[key].donationAmounts[msg.sender] = allDonations[key].donationAmounts[msg.sender].add(msg.value);
  projects[key].amountRaised = projects[key].amountRaised.add(msg.value);
  if (projectHasEnded(key)) {
    projects[key].projectIsOver = true;
    payRefunds(key);
  }
  if (fundingGoalReached(key)) {
    projects[key].projectIsOver = true;
    payOut(key);
  }
  return true;
}

// Checks is the projects end date has passed.
function projectHasEnded(uint key) public view returns (bool) {
  if (block.timestamp >= projects[key].projectEndTime) {
    return true;
  }
  return false;
}
```

This function takes a project key and adds the message value as a donation to the project with that key. It ensures the message sender is not also the project owner to prevent someone donating to their own project. It records the donators address, the amount donated and updates the amount raised for that project. It then checks if the project end date has been reached and if it has it pays refunds. It also checks if the funding goal has been reached and if so pays out to the project creator.

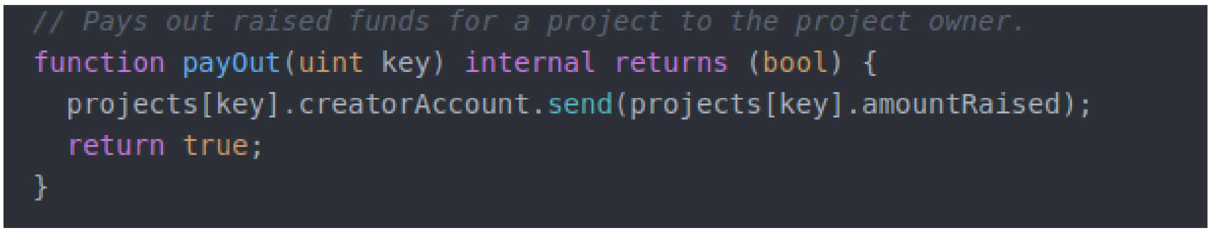
### 3.1.5. Check if a project has ended or the funding goal has been reached



```
// Checks is the projects end date has passed.
function projectHasEnded(uint key) public view returns (bool) {
  if (block.timestamp >= projects[key].projectEndTime) {
    return true;
  }
  return false;
}

// Checks if the funding goal has been reached.
function fundingGoalReached(uint key) public view returns (bool) {
  if (projects[key].amountRaised >= projects[key].fundingGoal) {
    return true;
  }
  return false;
}
```

These functions return a Boolean representing if the project has ended ether by the end date being reached or by the funding goal being achieved.

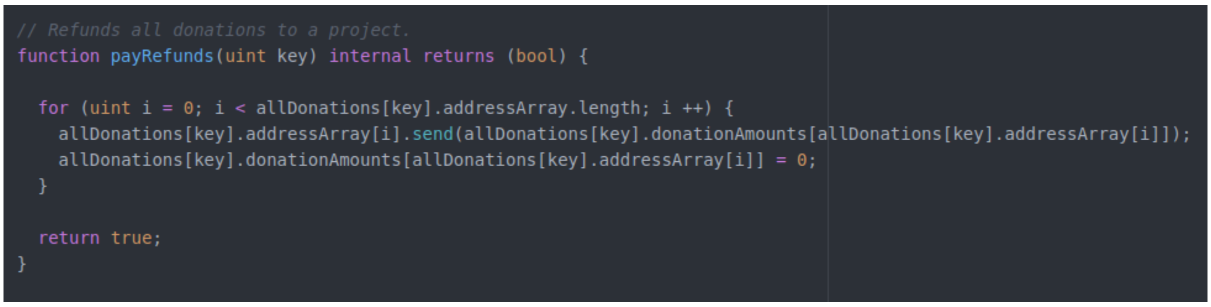
### 3.1.6. Pay-out to project creator



```
// Pays out raised funds for a project to the project owner.
function payOut(uint key) internal returns (bool) {
  projects[key].creatorAccount.send(projects[key].amountRaised);
  return true;
}
```

This function takes a projects key as input and pays out the amount raised for that project to the project's creator. This function can only be called internally for security reasons and is only called when a projects funding goal has been reached.

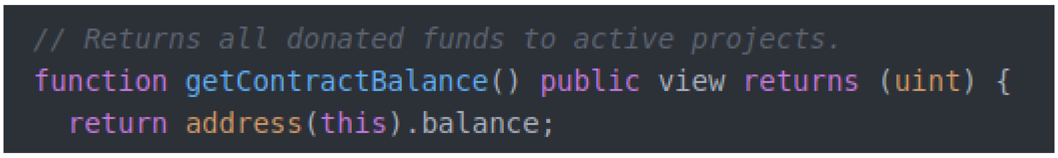
### 3.1.7. Pay refund to all projects donators



```
// Refunds all donations to a project.
function payRefunds(uint key) internal returns (bool) {

  for (uint i = 0; i < allDonations[key].addressArray.length; i ++) {
    allDonations[key].addressArray[i].send(allDonations[key].donationAmounts[allDonations[key].addressArray[i]]);
    allDonations[key].donationAmounts[allDonations[key].addressArray[i]] = 0;
  }

  return true;
}
```

This function takes a projects key as input and pays a refund to all accounts who donated to that project. It goes through the donation addresses array for that project and sends the amount they pledged. This function can only be called internally for security reasons and is only called when a projects end date has been reached.

### 3.1.8. Return contract balance



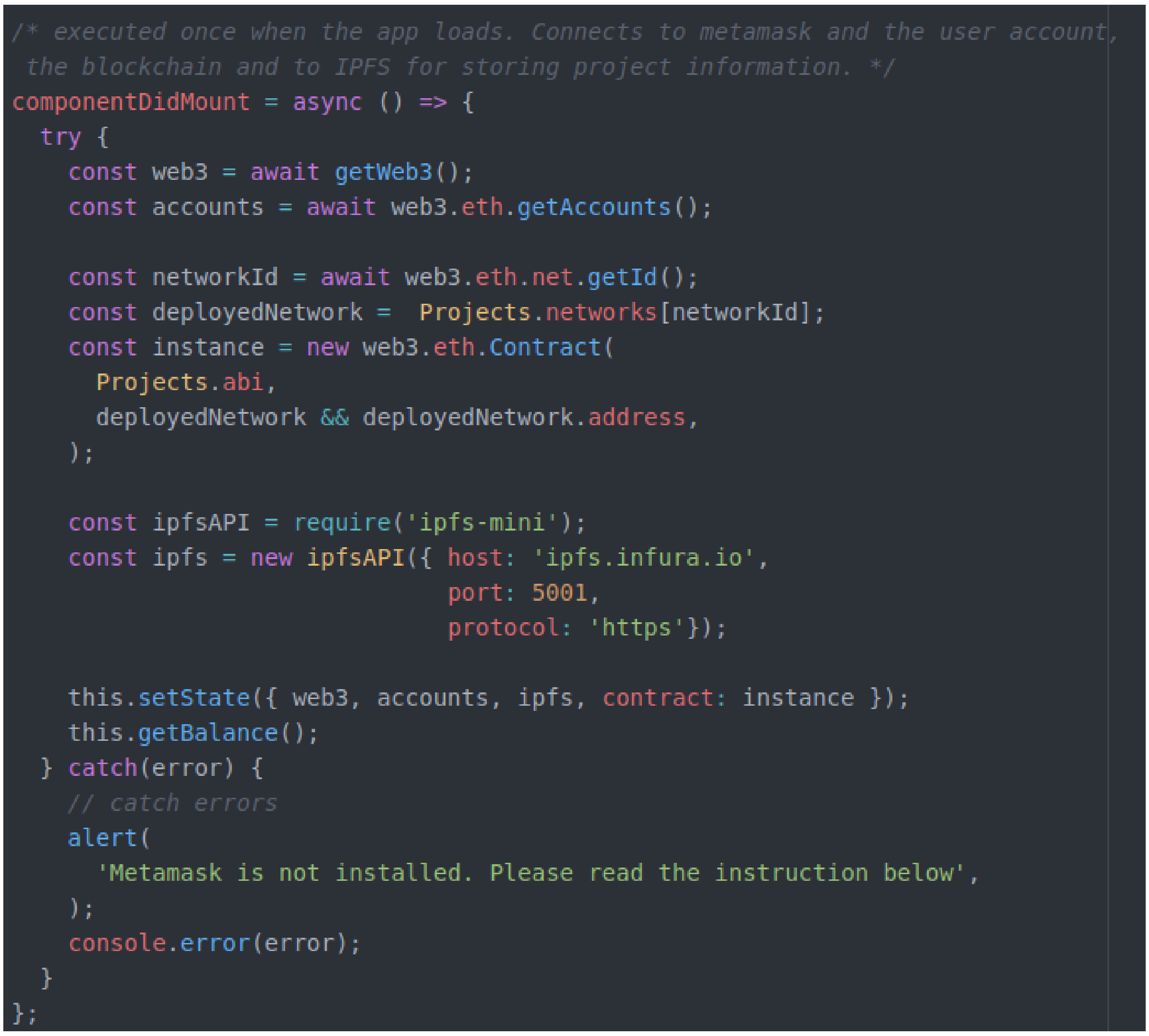
```
// Returns all donated funds to active projects.
function getContractBalance() public view returns (uint) {
  return address(this).balance;
```

This function just returns all funds being currently held in the contract. This represents all pledged donations to active projects. It is a view only function that does not cost gas to use.

## 3.2. User Interface Functions

This section explains the functions of the frontend application.

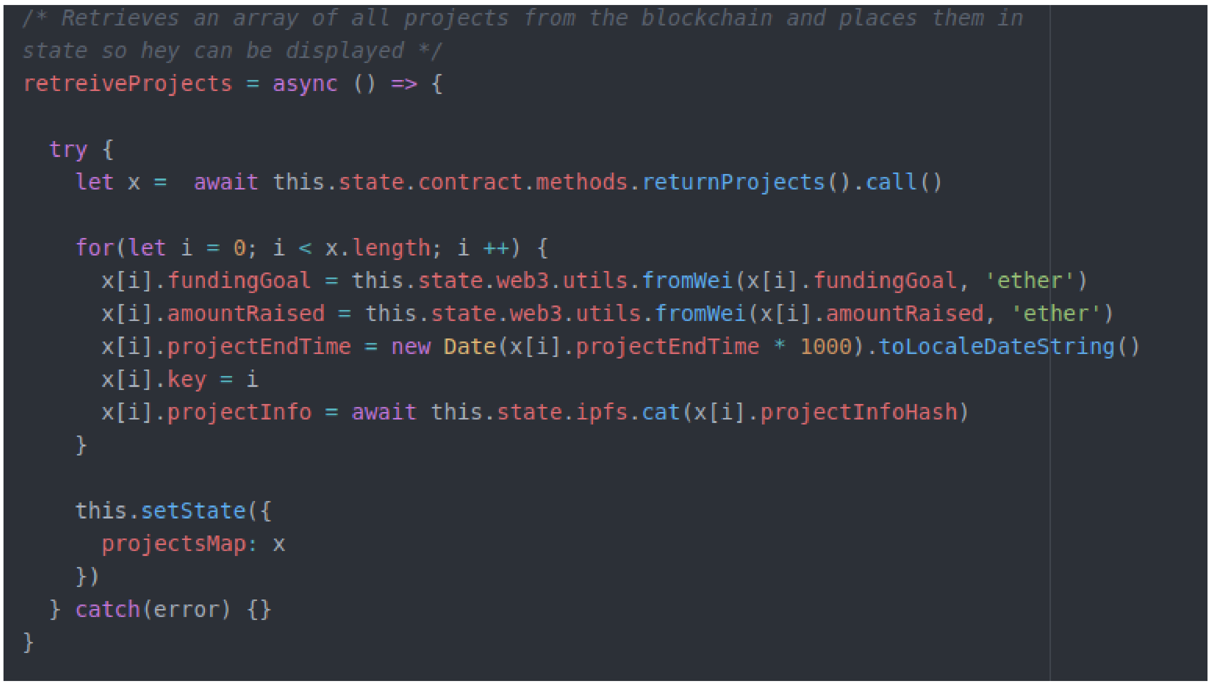
### 3.2.1. Connect to web3 and IPFS



```
/* executed once when the app loads. Connects to metamask and the user account,
  the blockchain and to IPFS for storing project information. */
componentDidMount = async () => {
  try {
    const web3 = await getWeb3();
    const accounts = await web3.eth.getAccounts();

    const networkId = await web3.eth.net.getId();
    const deployedNetwork =  Projects.networks[networkId];
    const instance = new web3.eth.Contract(
      Projects.abi,
      deployedNetwork && deployedNetwork.address,
    );

    const ipfsAPI = require('ipfs-mini');
    const ipfs = new ipfsAPI({ host: 'ipfs.infura.io',
                               port: 5001,
                               protocol: 'https'});

    this.setState({ web3, accounts, ipfs, contract: instance });
    this.getBalance();
  } catch(error) {
    // catch errors
    alert(
      'Metamask is not installed. Please read the instruction below',
    );
    console.error(error);
  }
};
```

componentDidMount is a special function in react that loads once when the app loads. Here we connect to web3, the deployed smart contract and IPFS and save these details into state so we can use them later. An error thrown here would mean that Metamask is not installed so we handle that with a catch and alert.

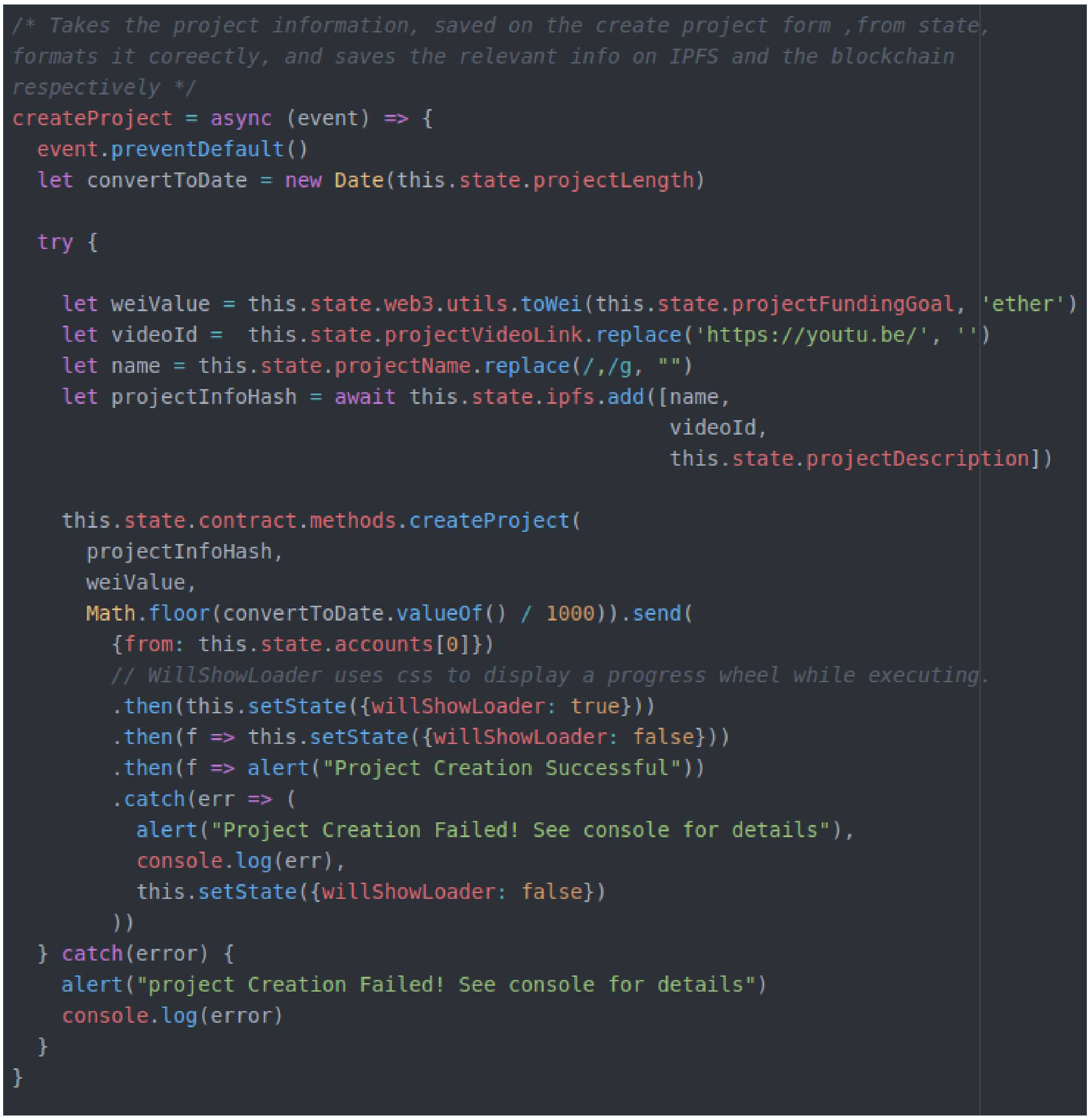
### 3.2.2. Retrieve project list from blockchain



```
/* Retrieves an array of all projects from the blockchain and places them in
state so hey can be displayed */
retreiveProjects = async () => {

  try {
    let x =  await this.state.contract.methods.returnProjects().call()

    for(let i = 0; i < x.length; i ++) {
      x[i].fundingGoal = this.state.web3.utils.fromWei(x[i].fundingGoal, 'ether')
      x[i].amountRaised = this.state.web3.utils.fromWei(x[i].amountRaised, 'ether')
      x[i].projectEndTime = new Date(x[i].projectEndTime * 1000).toLocaleDateString()
      x[i].key = i
      x[i].projectInfo = await this.state.ipfs.cat(x[i].projectInfoHash)
    }

    this.setState({
      projectsMap: x
    })
  } catch(error) {}
}
```

This function retrieves all projects from the blockchain and formats the information in a way it can be displayed on the view projects page of the user interface. For each project in the list, we must convert the amounts from wei to ether, convert the timestamps to a date string and look up IPFS with the hash to retrieve the other project info. When this is done, we save the project array into state.

### 3.2.3. Create a project

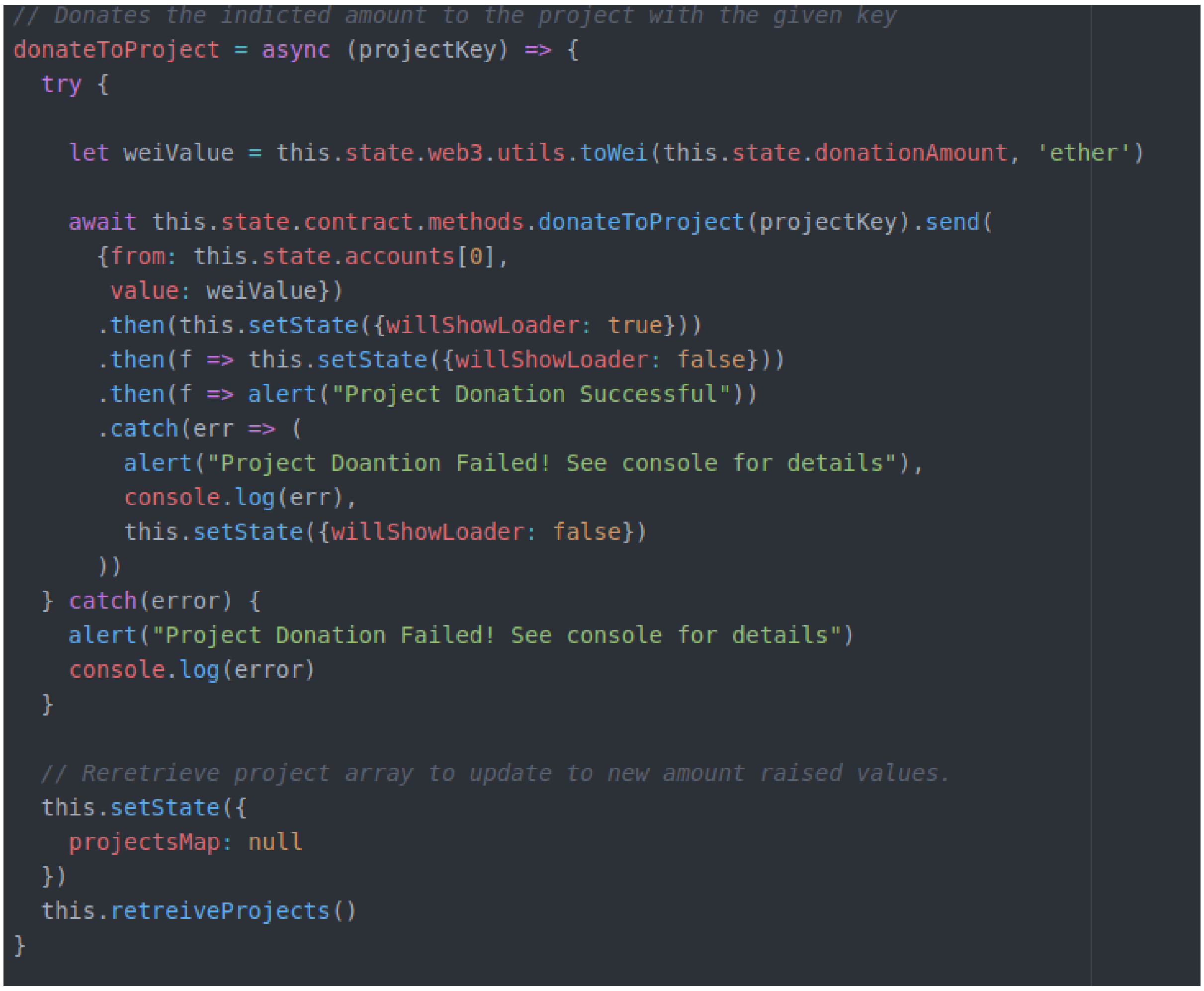


```
/* Takes the project information, saved on the create project form ,from state,
formats it coreectly, and saves the relevant info on IPFS and the blockchain
respectively */
createProject = async (event) => {
  event.preventDefault()
  let convertToDate = new Date(this.state.projectLength)

  try {

    let weiValue = this.state.web3.utils.toWei(this.state.projectFundingGoal, 'ether')
    let videoId =  this.state.projectVideoLink.replace('https://youtu.be/', '')
    let name = this.state.projectName.replace(/,/g, "")
    let projectInfoHash = await this.state.ipfs.add([name,
                                                     videoId,
                                                     this.state.projectDescription])

    this.state.contract.methods.createProject(
      projectInfoHash,
      weiValue,
      Math.floor(convertToDate.valueOf() / 1000)).send(
        {from: this.state.accounts[0]})
        // WillShowLoader uses css to display a progress wheel while executing.
        .then(this.setState({willShowLoader: true}))
        .then(f => this.setState({willShowLoader: false}))
        .then(f => alert("Project Creation Successful"))
        .catch(err => (
          alert("Project Creation Failed! See console for details"),
          console.log(err),
          this.setState({willShowLoader: false})
        ))
  } catch(error) {
    alert("project Creation Failed! See console for details")
    console.log(error)
  }
}
```

This function is called when a user submits a create project form. It adds a new project to the blockchain. The form values are taken from state and converted to the correct format for storing on the blockchain and to IPFS. We convert the Eth funding goal to wei, we strip the start off the video link which allows us to rebuild it later on as an embedded video link and we strip the commas from the projects name which helps fix a bug.

With this done we send to project name, video link and description to IPFS and save the returned hash. We then call the createProject function from the smart contract and send the hash, the funding goal and the end date as a timestamp to the blockchain. The willShowLoader attribute allows a loading wheel implemented with CSS to be shown as the block mines so the user has some feedback and knows what is happening. An alert lets the user know whether the project's creation has been successful or not.

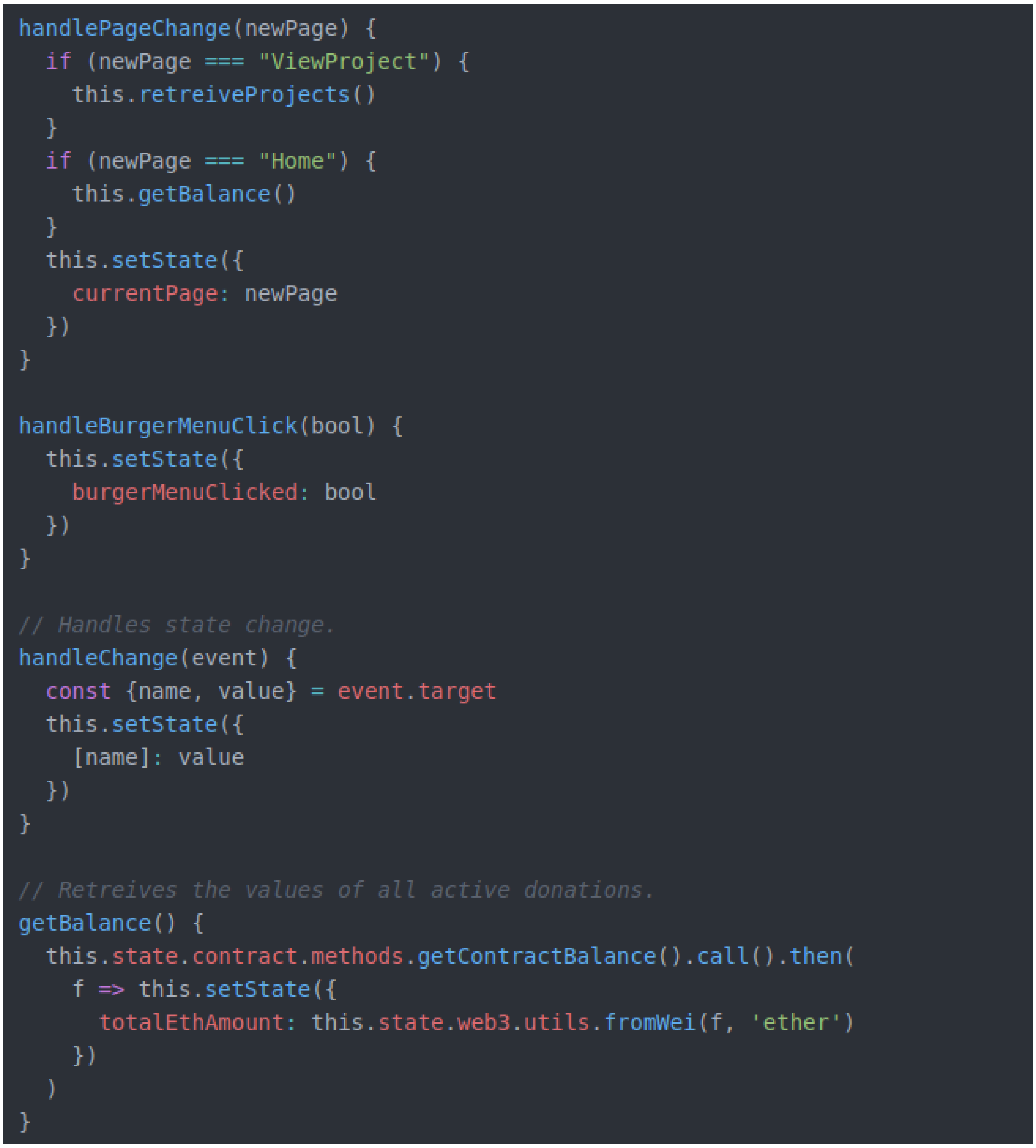
### 3.2.4. Donate to a project



```
// Donates the indicted amount to the project with the given key
donateToProject = async (projectKey) => {
  try {

    let weiValue = this.state.web3.utils.toWei(this.state.donationAmount, 'ether')

    await this.state.contract.methods.donateToProject(projectKey).send(
      {from: this.state.accounts[0],
       value: weiValue})
      .then(this.setState({willShowLoader: true}))
      .then(f => this.setState({willShowLoader: false}))
      .then(f => alert("Project Donation Successful"))
      .catch(err => (
        alert("Project Doantion Failed! See console for details"),
        console.log(err),
        this.setState({willShowLoader: false})
      ))
  } catch(error) {
    alert("Project Donation Failed! See console for details")
    console.log(error)
  }

  // Reretrieve project array to update to new amount raised values.
  this.setState({
    projectsMap: null
  })
  this.retreiveProjects()
}
```

This function is called when a user clicks the donate button on the view projects page. It converts the eth value to wei and then sends a message with that value from the donators address along with the project’s id number. The loading wheel works the same as in the create project function above and the user is informed by an alert whether the function has been successful. The projects map in state is then reset and re retrieved to update to the amount raised values.

### 3.2.5. Miscellaneous

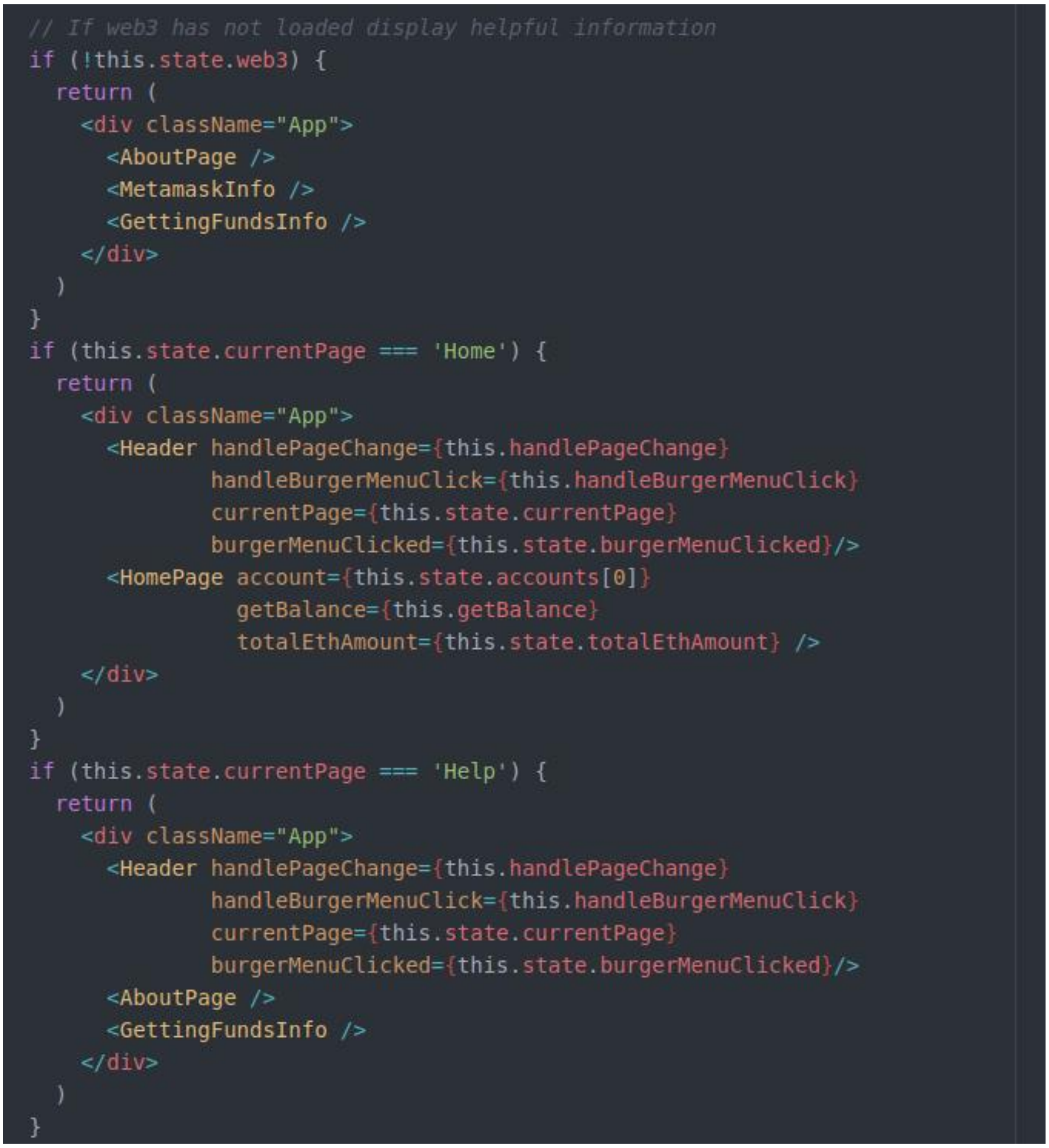


```
handlePageChange(newPage) {
  if (newPage === "ViewProject") {
    this.retreiveProjects()
  }
  if (newPage === "Home") {
    this.getBalance()
  }
  this.setState({
    currentPage: newPage
  })
}

handleBurgerMenuClick(bool) {
  this.setState({
    burgerMenuClicked: bool
  })
}

// Handles state change.
handleChange(event) {
  const {name, value} = event.target
  this.setState({
    [name]: value
  })
}

// Retreives the values of all active donations.
getBalance() {
  this.state.contract.methods.getContractBalance().call().then(
    f => this.setState({
      totalEthAmount: this.state.web3.utils.fromWei(f, 'ether')
    })
  )
}
```

The handle change functions handle state changes which mange things like page navigation, button clicks and form fields updating. The get balance function retrieves the amount raised for all active projects from the smart contract.

## 3.3. User Interface Components

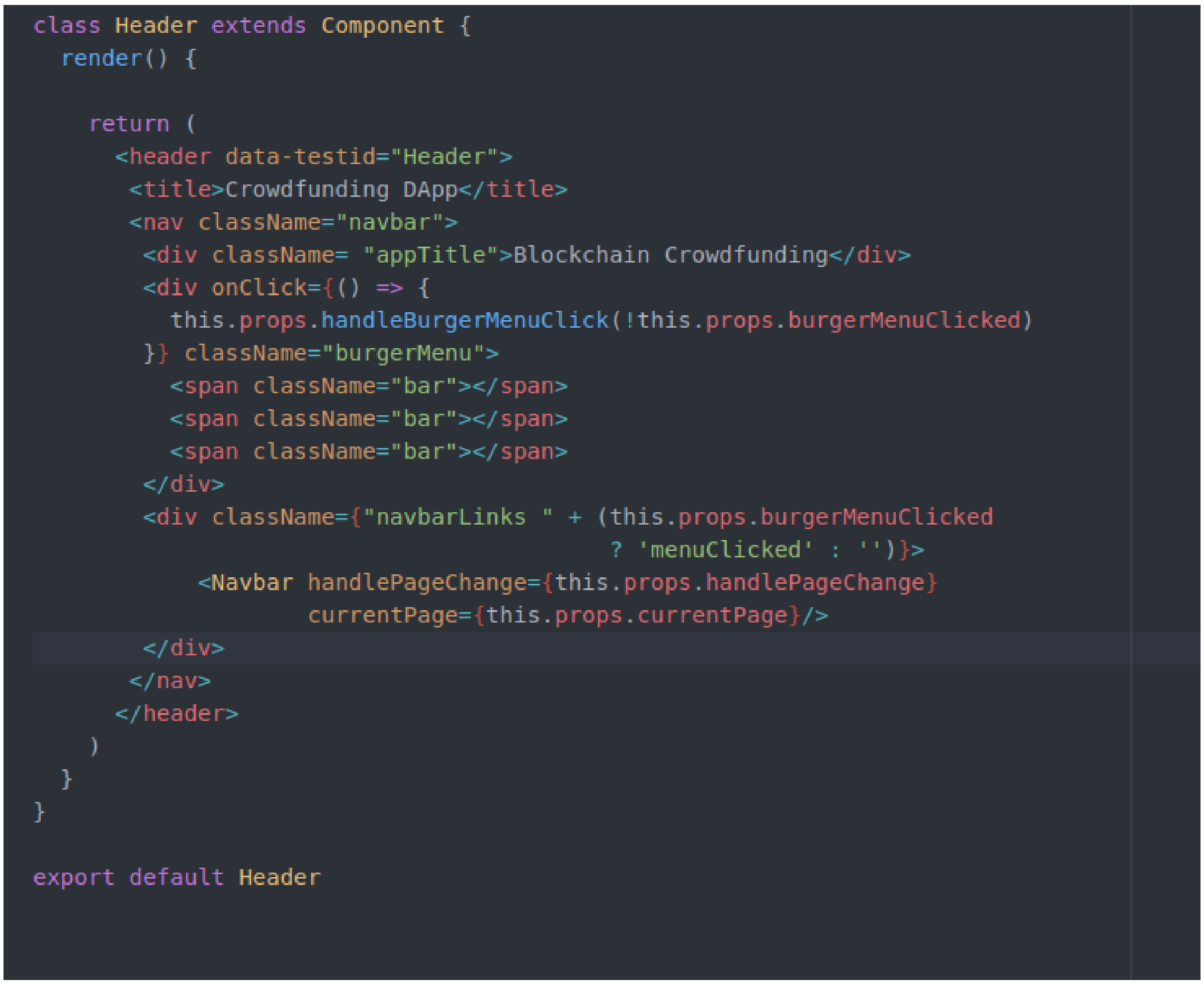
The concept of React user interfaces involves modular components that are rendered conditionally based on state. An example is shown below.



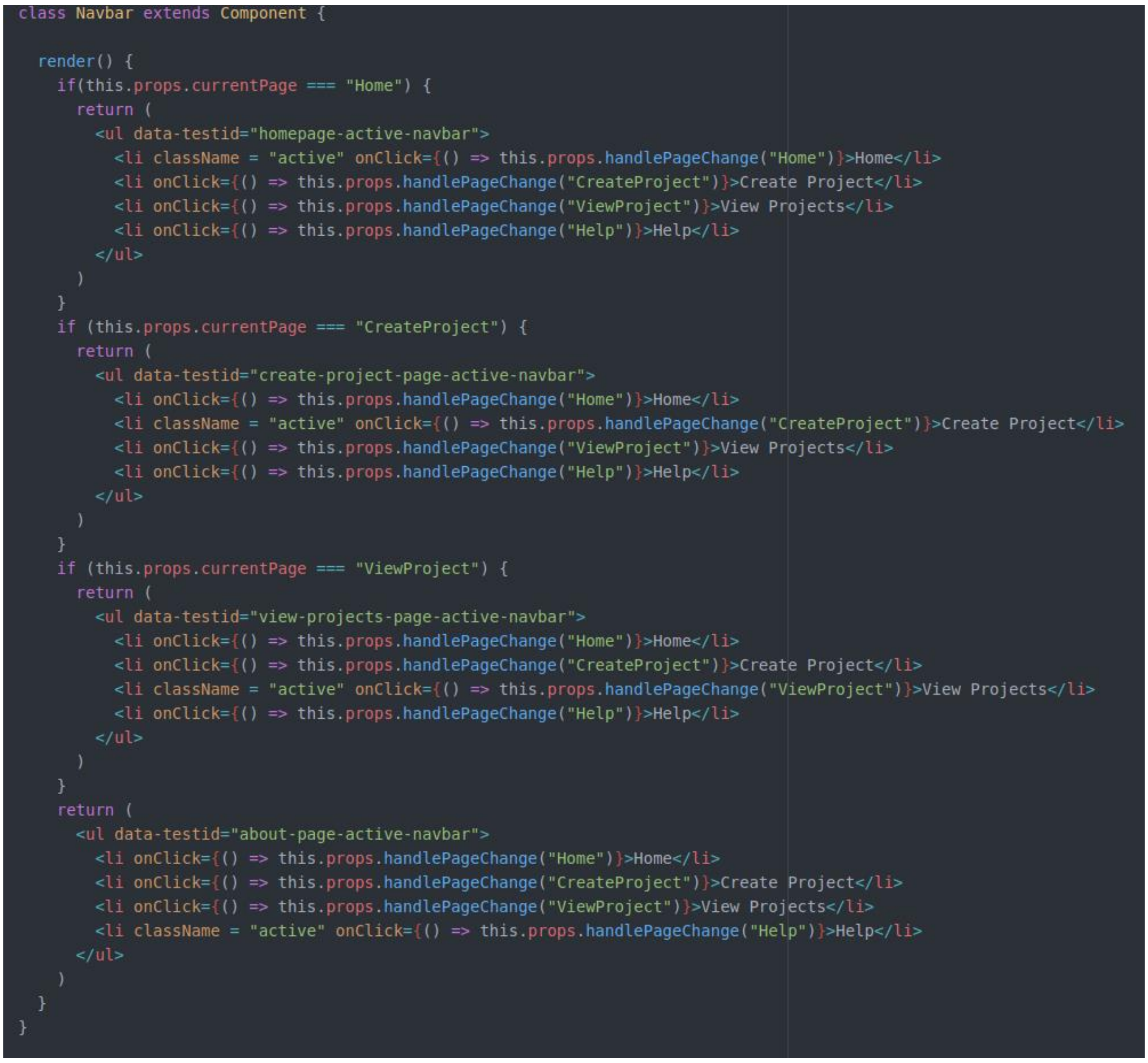
```
// If web3 has not loaded display helpful information
if (!this.state.web3) {
  return (
    <div className="App">
      <AboutPage />
      <MetamaskInfo />
      <GettingFundsInfo />
    </div>
  )
}
if (this.state.currentPage === 'Home') {
  return (
    <div className="App">
      <Header handlePageChange={this.handlePageChange}
              handleBurgerMenuClick={this.handleBurgerMenuClick}
              currentPage={this.state.currentPage}
              burgerMenuClicked={this.state.burgerMenuClicked}/>
      <HomePage account={this.state.accounts[0]}
                getBalance={this.getBalance}
                totalEthAmount={this.state.totalEthAmount} />
    </div>
  )
}
if (this.state.currentPage === 'Help') {
  return (
    <div className="App">
      <Header handlePageChange={this.handlePageChange}
              handleBurgerMenuClick={this.handleBurgerMenuClick}
              currentPage={this.state.currentPage}
              burgerMenuClicked={this.state.burgerMenuClicked}/>
      <AboutPage />
      <GettingFundsInfo />
    </div>
  )
}
```

There are too many components to list here and most, if not all of them, are self-explanatory do to their small modular nature. In this section I have listed some of the larger components

### 3.3.1. Header and Navbar



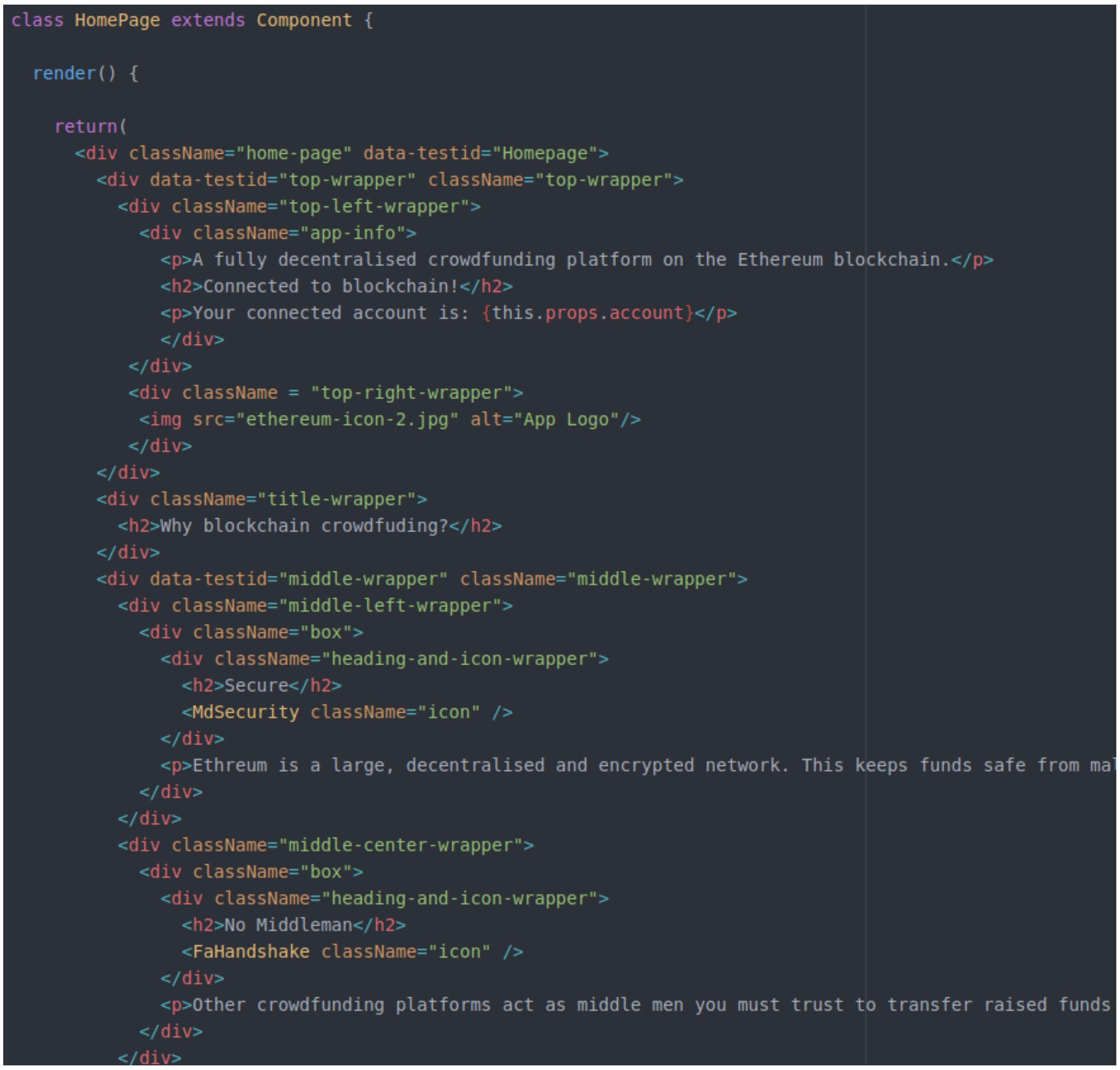
```
class Header extends Component {
  render() {

    return (
      <header data-testid="Header">
       <title>Crowdfunding DApp</title>
       <nav className="navbar">
        <div className= "appTitle">Blockchain Crowdfunding</div>
        <div onClick={() => {
          this.props.handleBurgerMenuClick(!this.props.burgerMenuClicked)
        }} className="burgerMenu">
          <span className="bar"></span>
          <span className="bar"></span>
          <span className="bar"></span>
        </div>
        <div className={"navbarLinks " + (this.props.burgerMenuClicked
                                           ? 'menuClicked' : '')}>
            <Navbar handlePageChange={this.props.handlePageChange}
                    currentPage={this.props.currentPage}/>
        </div>
       </nav>
      </header>
    )
  }
}

export default Header
```



```
class Navbar extends Component {

  render() {
    if(this.props.currentPage === "Home") {
      return (
        <ul data-testid="homepage-active-navbar">
          <li className = "active" onClick={() => this.props.handlePageChange("Home")}>Home</li>
          <li onClick={() => this.props.handlePageChange("CreateProject")}>Create Project</li>
          <li onClick={() => this.props.handlePageChange("ViewProject")}>View Projects</li>
          <li onClick={() => this.props.handlePageChange("Help")}>Help</li>
        </ul>
      )
    }
    if (this.props.currentPage === "CreateProject") {
      return (
        <ul data-testid="create-project-page-active-navbar">
          <li onClick={() => this.props.handlePageChange("Home")}>Home</li>
          <li className = "active" onClick={() => this.props.handlePageChange("CreateProject")}>Create Project</li>
          <li onClick={() => this.props.handlePageChange("ViewProject")}>View Projects</li>
          <li onClick={() => this.props.handlePageChange("Help")}>Help</li>
        </ul>
      )
    }
    if (this.props.currentPage === "ViewProject") {
      return (
        <ul data-testid="view-projects-page-active-navbar">
          <li onClick={() => this.props.handlePageChange("Home")}>Home</li>
          <li onClick={() => this.props.handlePageChange("CreateProject")}>Create Project</li>
          <li className = "active" onClick={() => this.props.handlePageChange("ViewProject")}>View Projects</li>
          <li onClick={() => this.props.handlePageChange("Help")}>Help</li>
        </ul>
      )
    }
    return (
      <ul data-testid="about-page-active-navbar">
        <li onClick={() => this.props.handlePageChange("Home")}>Home</li>
        <li onClick={() => this.props.handlePageChange("CreateProject")}>Create Project</li>
        <li onClick={() => this.props.handlePageChange("ViewProject")}>View Projects</li>
        <li className = "active" onClick={() => this.props.handlePageChange("Help")}>Help</li>
      </ul>
    )
  }
}
```

These are the header and navbar components. The header component renders the App title, the navbar and the burger menu navbar. The navbar handles the functionality of page navigation and marks the active page for highlighting with CSS. The burger menu navbar is only displayed when the screen is too narrow to display the full navbar, (for example on mobile), this is handled with CSS.

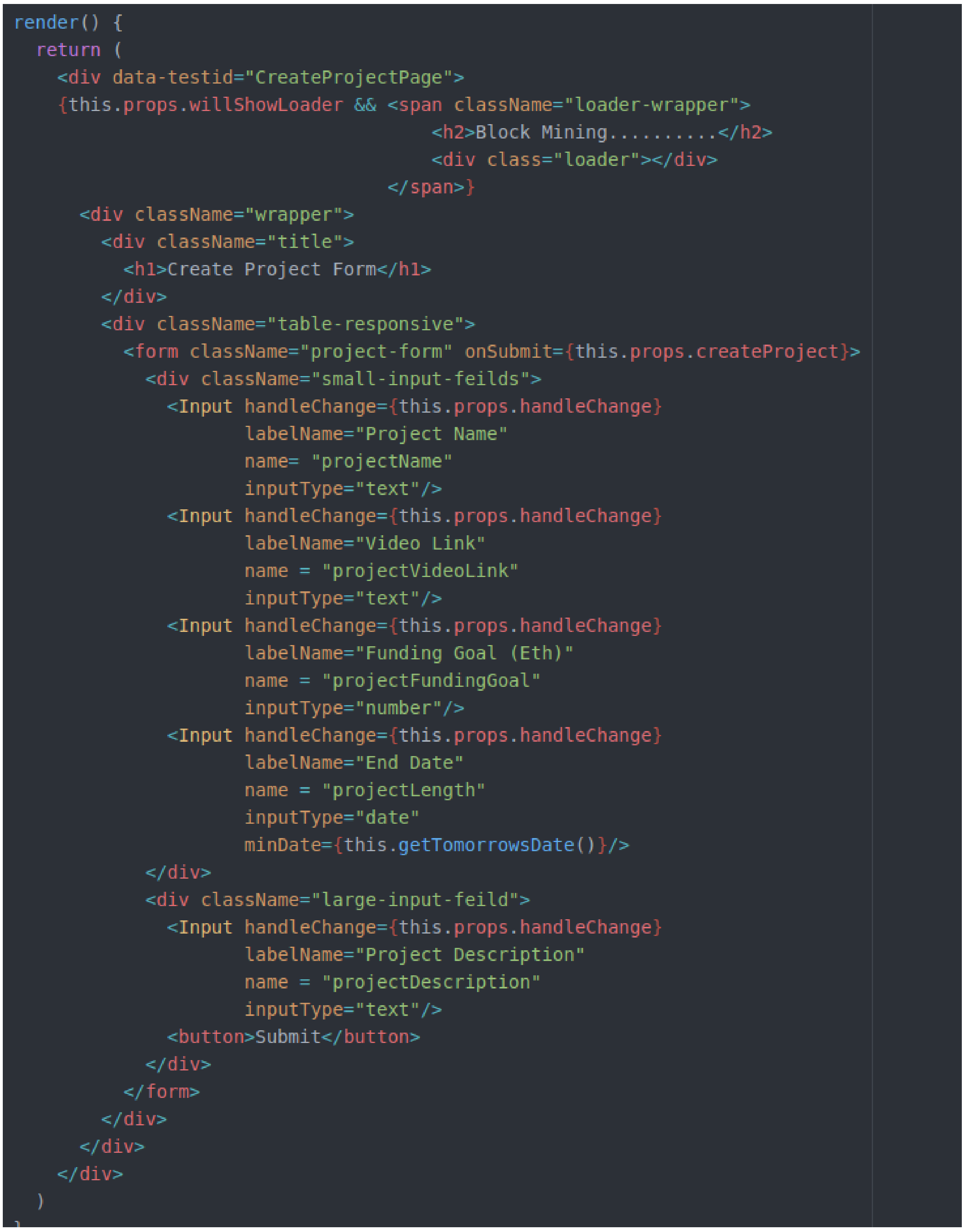
### 3.3.2. Homepage



```
class HomePage extends Component {

  render() {

    return(
      <div className="home-page" data-testid="Homepage">
        <div data-testid="top-wrapper" className="top-wrapper">
          <div className="top-left-wrapper">
            <div className="app-info">
              <p>A fully decentralised crowdfunding platform on the Ethereum blockchain.</p>
              <h2>Connected to blockchain!</h2>
              <p>Your connected account is: {this.props.account}</p>
              </div>
           </div>
           <div className = "top-right-wrapper">
            <img src="ethereum-icon-2.jpg" alt="App Logo"/>
           </div>
        </div>
        <div className="title-wrapper">
          <h2>Why blockchain crowdfuding?</h2>
        </div>
        <div data-testid="middle-wrapper" className="middle-wrapper">
          <div className="middle-left-wrapper">
            <div className="box">
              <div className="heading-and-icon-wrapper">
                <h2>Secure</h2>
                <MdSecurity className="icon" />
              </div>
              <p>Ethreum is a large, decentralised and encrypted network. This keeps funds safe from ma
            </div>
          </div>
          <div className="middle-center-wrapper">
            <div className="box">
              <div className="heading-and-icon-wrapper">
                <h2>No Middleman</h2>
                <FaHandshake className="icon" />
              </div>
              <p>Other crowdfunding platforms act as middle men you must trust to transfer raised funds
            </div>
          </div>
```

This component is displayed after you log into Metamask to use the App. It contains a small amount of info including the users connected Metamask account number. The multiple divs and icon components are for styling with CSS.

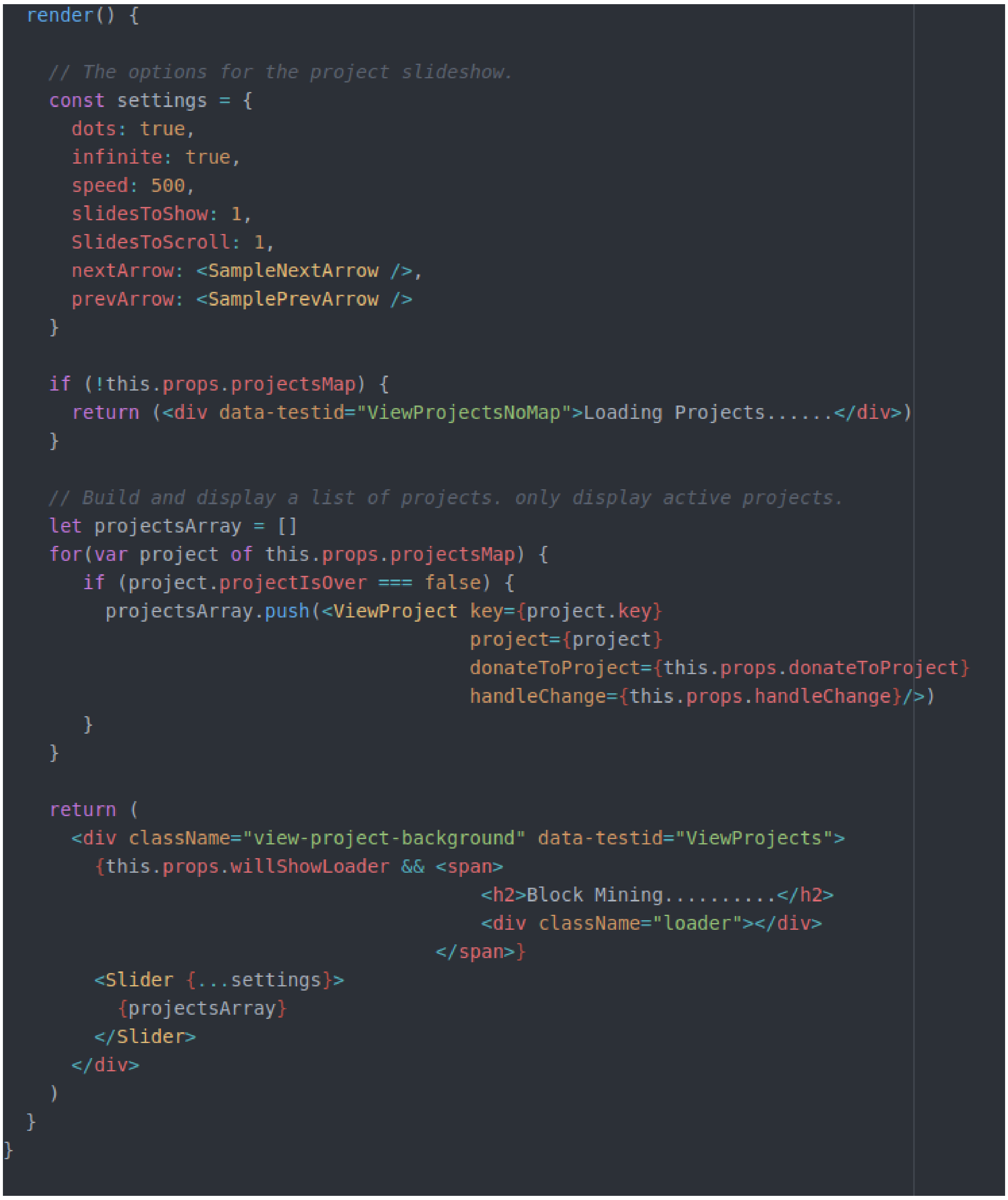
## 3.3.3. Create Project Page



```
render() {
  return (
    <div data-testid="CreateProjectPage">
    {this.props.willShowLoader && <span className="loader-wrapper">
                                      <h2>Block Mining..........</h2>
                                      <div class="loader"></div>
                                  </span>}
      <div className="wrapper">
        <div className="title">
          <h1>Create Project Form</h1>
        </div>
        <div className="table-responsive">
          <form className="project-form" onSubmit={this.props.createProject}>
            <div className="small-input-feilds">
              <Input handleChange={this.props.handleChange}
                     labelName="Project Name"
                     name= "projectName"
                     inputType="text"/>
              <Input handleChange={this.props.handleChange}
                     labelName="Video Link"
                     name = "projectVideoLink"
                     inputType="text"/>
              <Input handleChange={this.props.handleChange}
                     labelName="Funding Goal (Eth)"
                     name = "projectFundingGoal"
                     inputType="number"/>
              <Input handleChange={this.props.handleChange}
                     labelName="End Date"
                     name = "projectLength"
                     inputType="date"
                     minDate={this.getTomorrowsDate()}/>
            </div>
            <div className="large-input-feild">
              <Input handleChange={this.props.handleChange}
                     labelName="Project Description"
                     name = "projectDescription"
                     inputType="text"/>
              <button>Submit</button>
            </div>
          </form>
        </div>
      </div>
    </div>
  )
```

The create project page renders a form with various input fields and a submit button that calls the create project function. The input fields are individual input components which are explained later in this section. The loader wrapper class displays a CSS styled loading wheel. This is only shown when the willShowLoader prop is true. This is called during a block mining after the create project function is called.

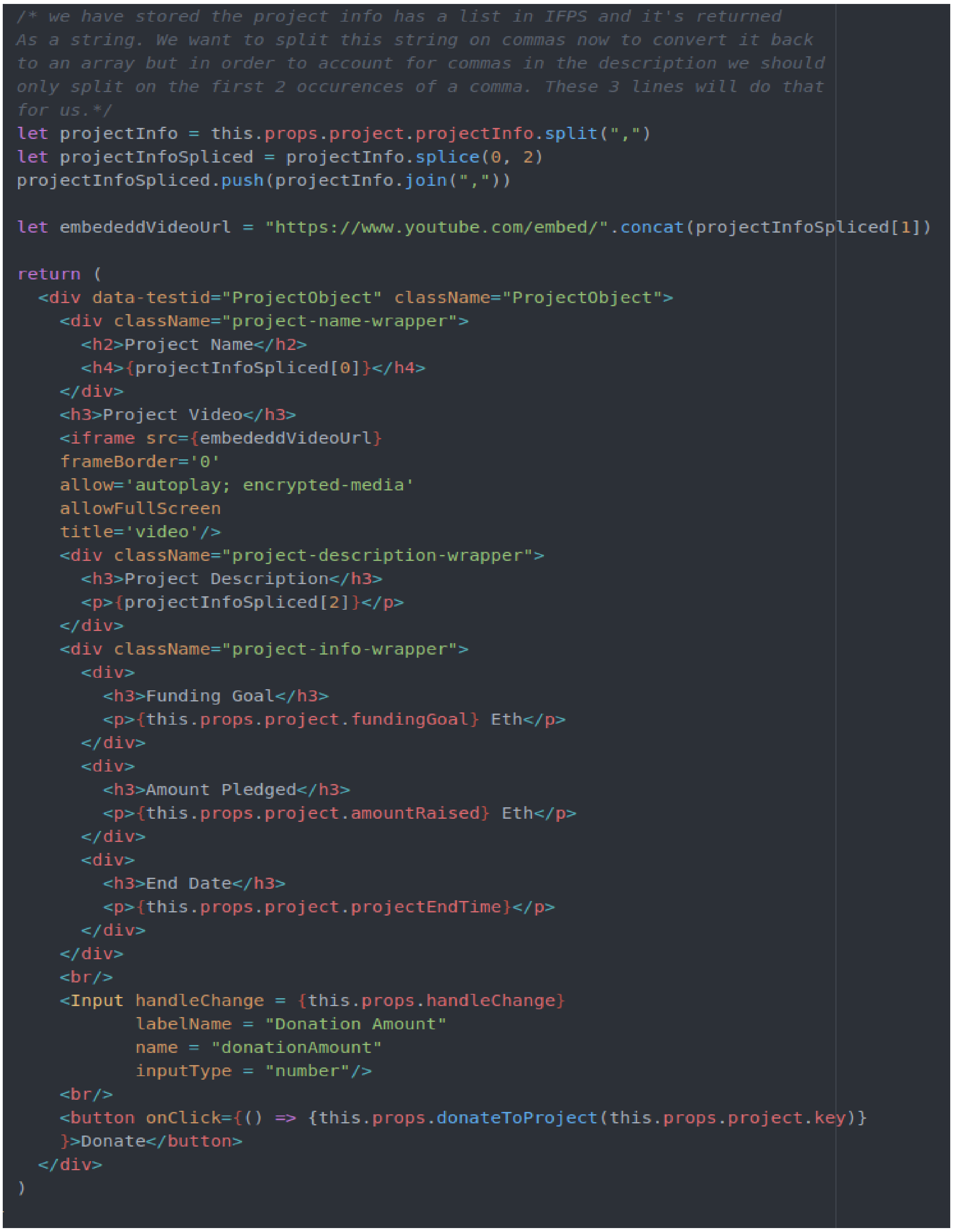
### 3.3.4. View Projects Page



```
  render() {

    // The options for the project slideshow.
    const settings = {
      dots: true,
      infinite: true,
      speed: 500,
      slidesToShow: 1,
      SlidesToScroll: 1,
      nextArrow: <SampleNextArrow />,
      prevArrow: <SamplePrevArrow />
    }

    if (!this.props.projectsMap) {
      return (<div data-testid="ViewProjectsNoMap">Loading Projects......</div>)
    }

    // Build and display a list of projects. only display active projects.
    let projectsArray = []
    for(var project of this.props.projectsMap) {
       if (project.projectIsOver === false) {
         projectsArray.push(<ViewProject key={project.key}
                                         project={project}
                                         donateToProject={this.props.donateToProject}
                                         handleChange={this.props.handleChange}/>)
       }
    }

    return (
      <div className="view-project-background" data-testid="ViewProjects">
        {this.props.willShowLoader && <span>
                                          <h2>Block Mining..........</h2>
                                          <div className="loader"></div>
                                        </span>}
        <Slider {...settings}>
          {projectsArray}
        </Slider>
      </div>
    )
  }
}
```

The view projects page takes the list of projects saved in state and displays them so a user can browse through them. It uses a Slider component which is a third-party library to display projects in a slideshow format. The settings variable applies some options to this slider component. This slider component is passed an array of view project components. When the array is built, we only add the active projects. The view project component is explained below.

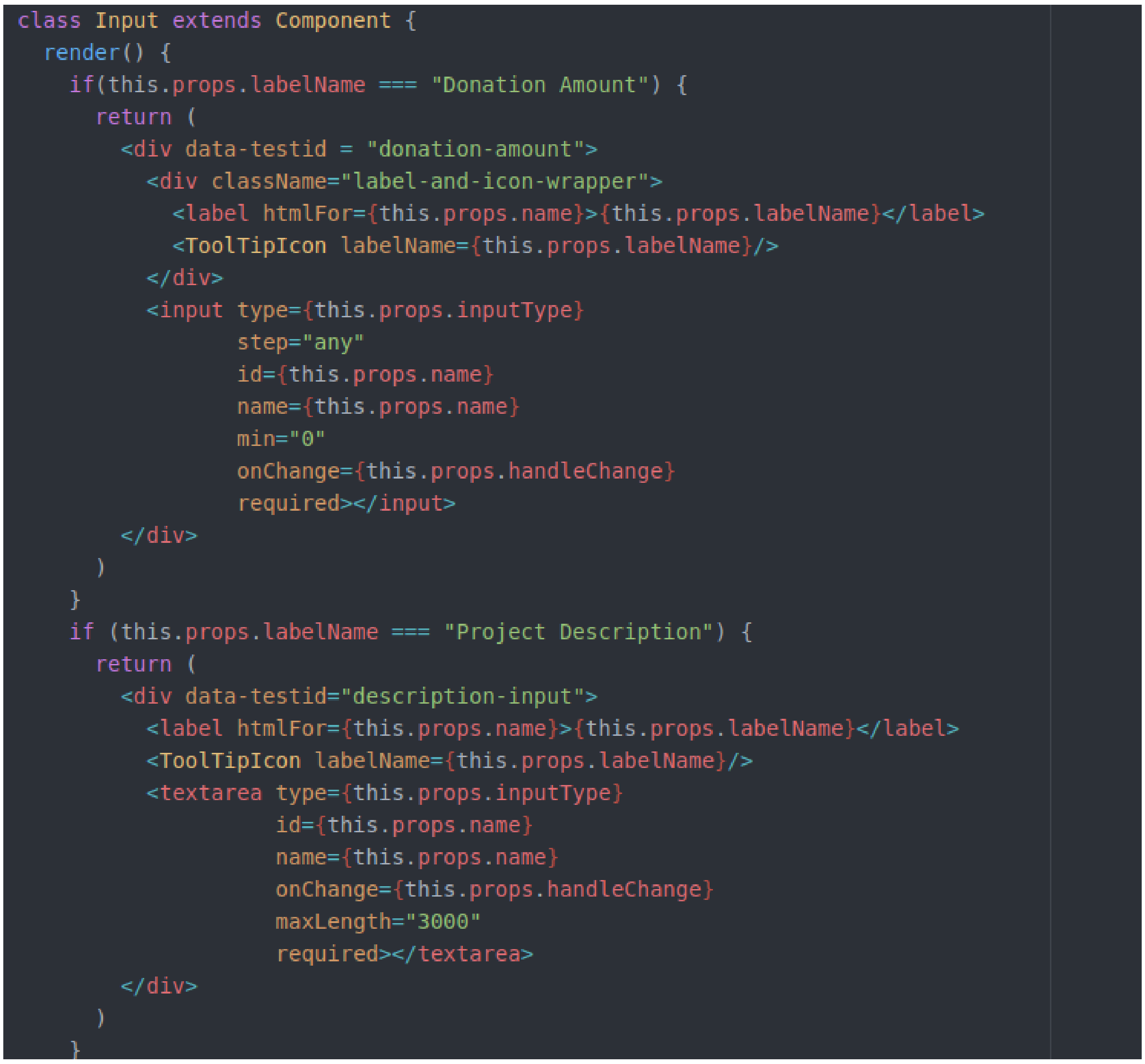
### 3.3.5. View Project



```
/* we have stored the project info has a list in IFPS and it's returned
As a string. We want to split this string on commas now to convert it back
to an array but in order to account for commas in the description we should
only split on the first 2 occurences of a comma. These 3 lines will do that
for us.*/
let projectInfo = this.props.project.projectInfo.split(",")
let projectInfoSpliced = projectInfo.splice(0, 2)
projectInfoSpliced.push(projectInfo.join(","))

let embededdVideoUrl = "https://www.youtube.com/embed/".concat(projectInfoSpliced[1])

return (
  <div data-testid="ProjectObject" className="ProjectObject">
    <div className="project-name-wrapper">
      <h2>Project Name</h2>
      <h4>{projectInfoSpliced[0]}</h4>
    </div>
    <h3>Project Video</h3>
    <iframe src={embededdVideoUrl}
    frameBorder='0'
    allow='autoplay; encrypted-media'
    allowFullScreen
    title='video'/>
    <div className="project-description-wrapper">
      <h3>Project Description</h3>
      <p>{projectInfoSpliced[2]}</p>
    </div>
    <div className="project-info-wrapper">
      <div>
        <h3>Funding Goal</h3>
        <p>{this.props.project.fundingGoal} Eth</p>
      </div>
      <div>
        <h3>Amount Pledged</h3>
        <p>{this.props.project.amountRaised} Eth</p>
      </div>
      <div>
        <h3>End Date</h3>
        <p>{this.props.project.projectEndTime}</p>
      </div>
    </div>
    <br/>
    <Input handleChange = {this.props.handleChange}
           labelName = "Donation Amount"
           name = "donationAmount"
           inputType = "number"/>
    <br/>
    <button onClick={() => {this.props.donateToProject(this.props.project.key)}
    }>Donate</button>
  </div>
)
```

This component renders one project in the view projects list. The project info saved to IPFS as an array is returned as a string so before rendering, we need to convert it back to an array. We also want to build an embedded YouTube link from our project video so it can be displayed in the page. The component then renders the project information including an iframe for the video, an input field for a donation amount and a button to donate to the project.

### 3.3.6. Input



```
class Input extends Component {
  render() {
    if(this.props.labelName === "Donation Amount") {
      return (
        <div data-testid = "donation-amount">
          <div className="label-and-icon-wrapper">
            <label htmlFor={this.props.name}>{this.props.labelName}</label>
            <ToolTipIcon labelName={this.props.labelName}/>
          </div>
          <input type={this.props.inputType}
                 step="any"
                 id={this.props.name}
                 name={this.props.name}
                 min="0"
                 onChange={this.props.handleChange}
                 required></input>
        </div>
      )
    }
    if (this.props.labelName === "Project Description") {
      return (
        <div data-testid="description-input">
          <label htmlFor={this.props.name}>{this.props.labelName}</label>
          <ToolTipIcon labelName={this.props.labelName}/>
          <textarea type={this.props.inputType}
                    id={this.props.name}
                    name={this.props.name}
                    onChange={this.props.handleChange}
                    maxLength="3000"
                    required></textarea>
        </div>
      )
    }
```

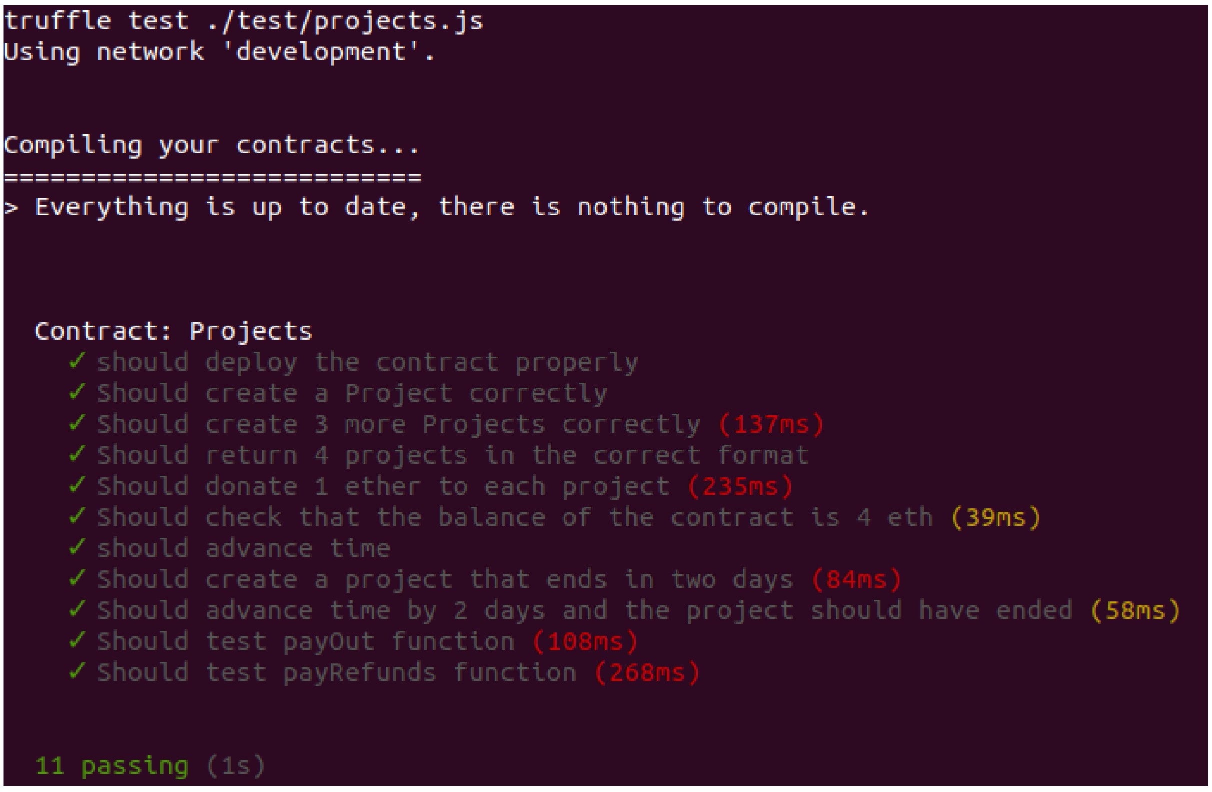
This is the input component that renders when there is an input field. Depending on the label passed as props the correct type of field will be rendered including the correct tool tip icon. For example, for a donation amount the input type is a number, the min is 0 and the step attribute allows decimal. For a project description the input type is a text area with a large max length.

# 4. Testing

This section contains a detailed description of the testing strategy and the tests performed. I have broken it down into 4 sections.

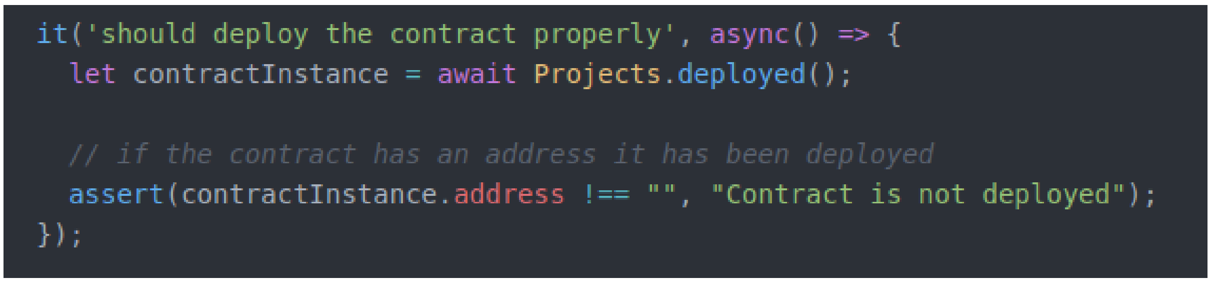
## 4.1. Smart contract unit and end-to-end testing

These tests are written in JavaScript and run with truffles built in testing framework. The tests are in the projects.js file in /src/test. To run the tests, you must have a local Ganache blockchain running on port 8546. These tests are unit tests for the specific functions in the smart contract as well as end-to-end tests which tests every aspect of functionality from a project being created to a project ending. Example of output when running these tests is seen below.



```
truffle test ./test/projects.js
Using network 'development'.


Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



  Contract: Projects
    ✓ should deploy the contract properly
    ✓ Should create a Project correctly
    ✓ Should create 3 more Projects correctly (137ms)
    ✓ Should return 4 projects in the correct format
    ✓ Should donate 1 ether to each project (235ms)
    ✓ Should check that the balance of the contract is 4 eth (39ms)
    ✓ should advance time
    ✓ Should create a project that ends in two days (84ms)
    ✓ Should advance time by 2 days and the project should have ended (58ms)
    ✓ Should test payOut function (108ms)
    ✓ Should test payRefunds function (268ms)


  11 passing (1s)
```

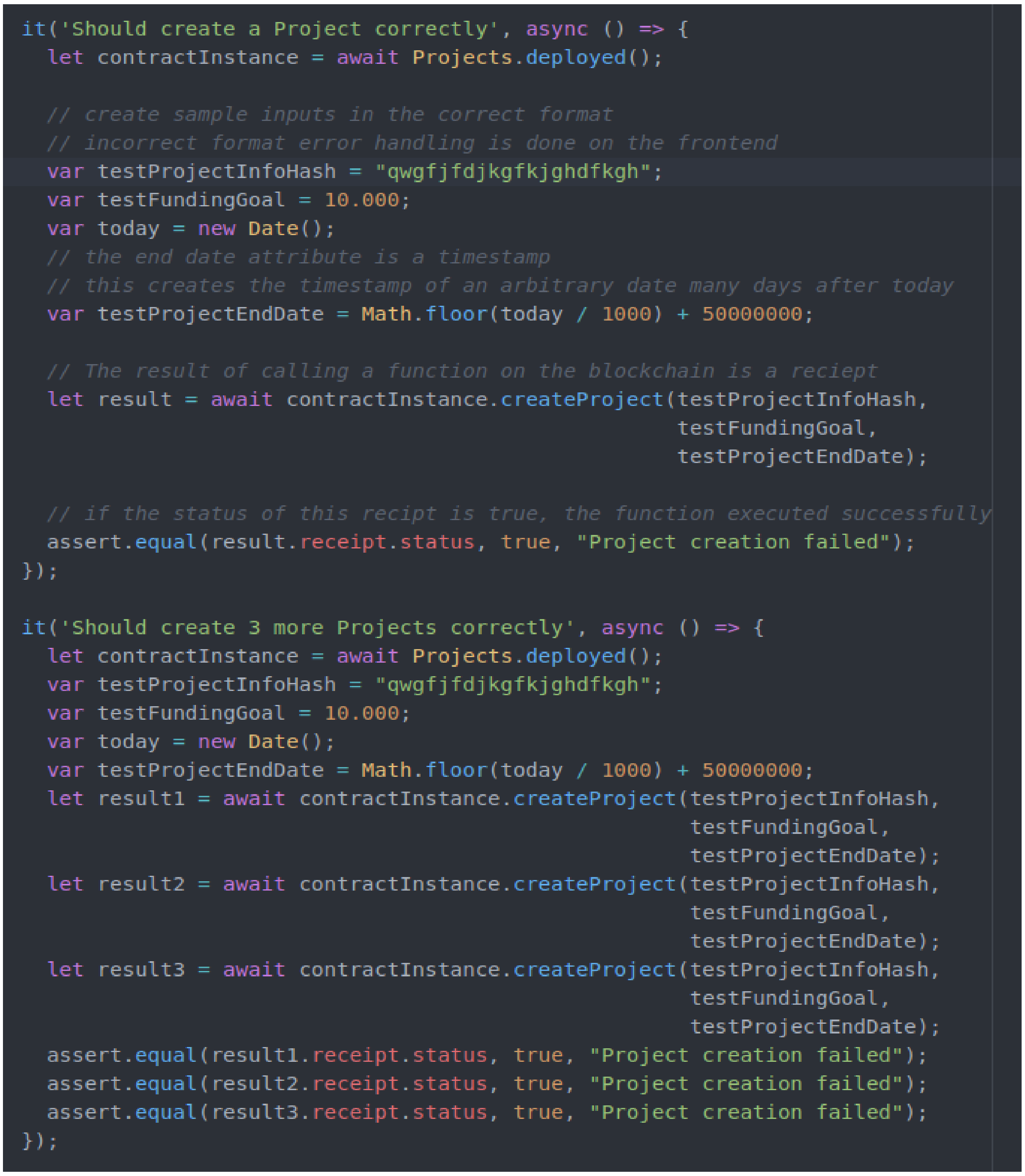
### 4.1.1. Testing contract deployment



```
it('should deploy the contract properly', async() => {
  let contractInstance = await Projects.deployed();

  // if the contract has an address it has been deployed
  assert(contractInstance.address !== "", "Contract is not deployed");
});
```

This test just ensures that the contract is deploying properly by checking if it has an address on the blockchain.

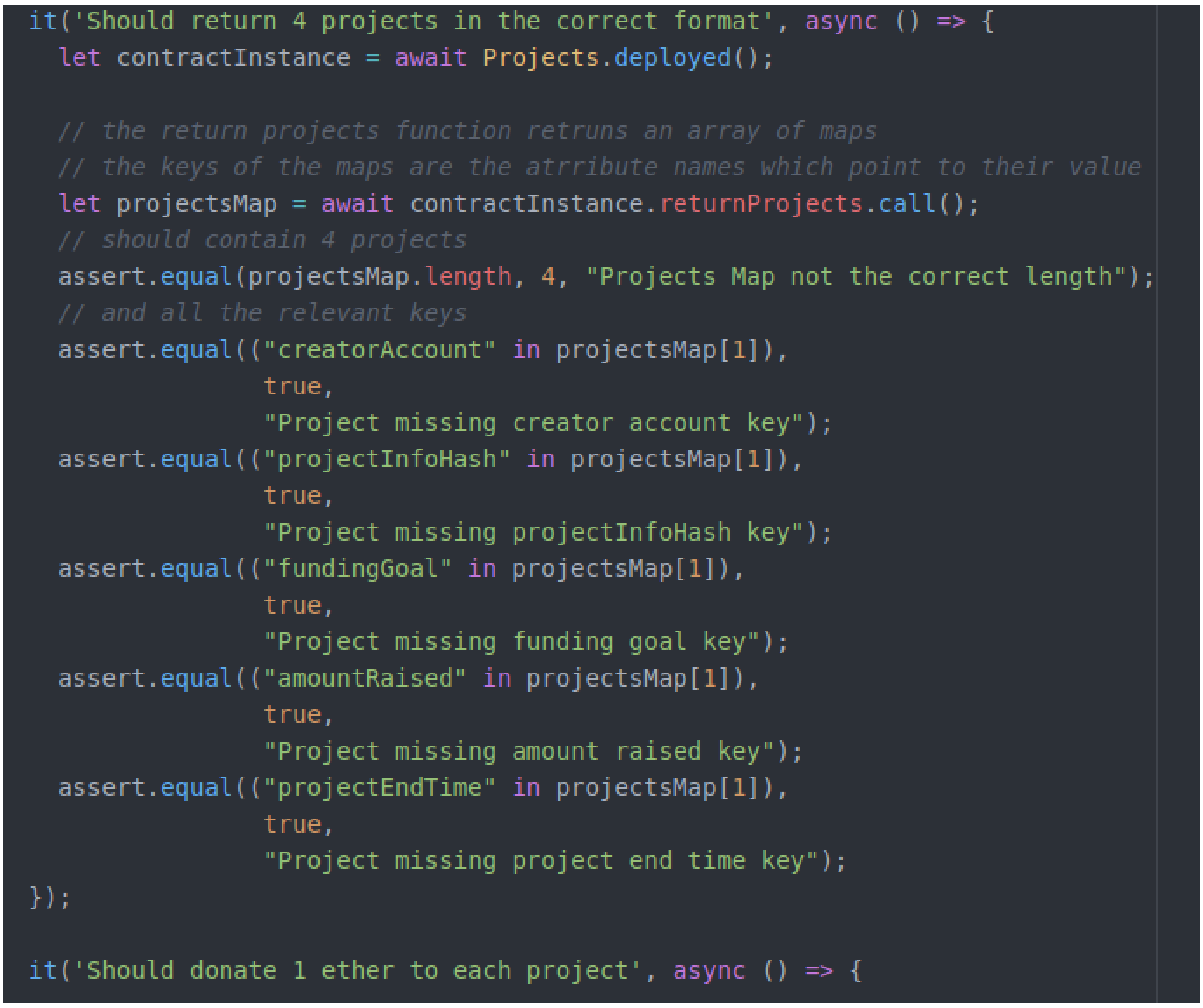
### 4.1.2. Testing contract creation



```
it('Should create a Project correctly', async () => {
  let contractInstance = await Projects.deployed();

  // create sample inputs in the correct format
  // incorrect format error handling is done on the frontend
  var testProjectInfoHash = "qwgfjfdjkgfkjghdfkgh";
  var testFundingGoal = 10.000;
  var today = new Date();
  // the end date attribute is a timestamp
  // this creates the timestamp of an arbitrary date many days after today
  var testProjectEndDate = Math.floor(today / 1000) + 50000000;

  // The result of calling a function on the blockchain is a reciept
  let result = await contractInstance.createProject(testProjectInfoHash,
                                                    testFundingGoal,
                                                    testProjectEndDate);

  // if the status of this recipt is true, the function executed successfully
  assert.equal(result.receipt.status, true, "Project creation failed");
});

it('Should create 3 more Projects correctly', async () => {
  let contractInstance = await Projects.deployed();
  var testProjectInfoHash = "qwgfjfdjkgfkjghdfkgh";
  var testFundingGoal = 10.000;
  var today = new Date();
  var testProjectEndDate = Math.floor(today / 1000) + 50000000;
  let result1 = await contractInstance.createProject(testProjectInfoHash,
                                                     testFundingGoal,
                                                     testProjectEndDate);
  let result2 = await contractInstance.createProject(testProjectInfoHash,
                                                     testFundingGoal,
                                                     testProjectEndDate);
  let result3 = await contractInstance.createProject(testProjectInfoHash,
                                                     testFundingGoal,
                                                     testProjectEndDate);
  assert.equal(result1.receipt.status, true, "Project creation failed");
  assert.equal(result2.receipt.status, true, "Project creation failed");
  assert.equal(result3.receipt.status, true, "Project creation failed");
});
```

These tests use placeholder sample input and call the create project function for 1 project and 3 projects, respectively. It then checks the status of the transaction receipts to ensure that project creation was successful.

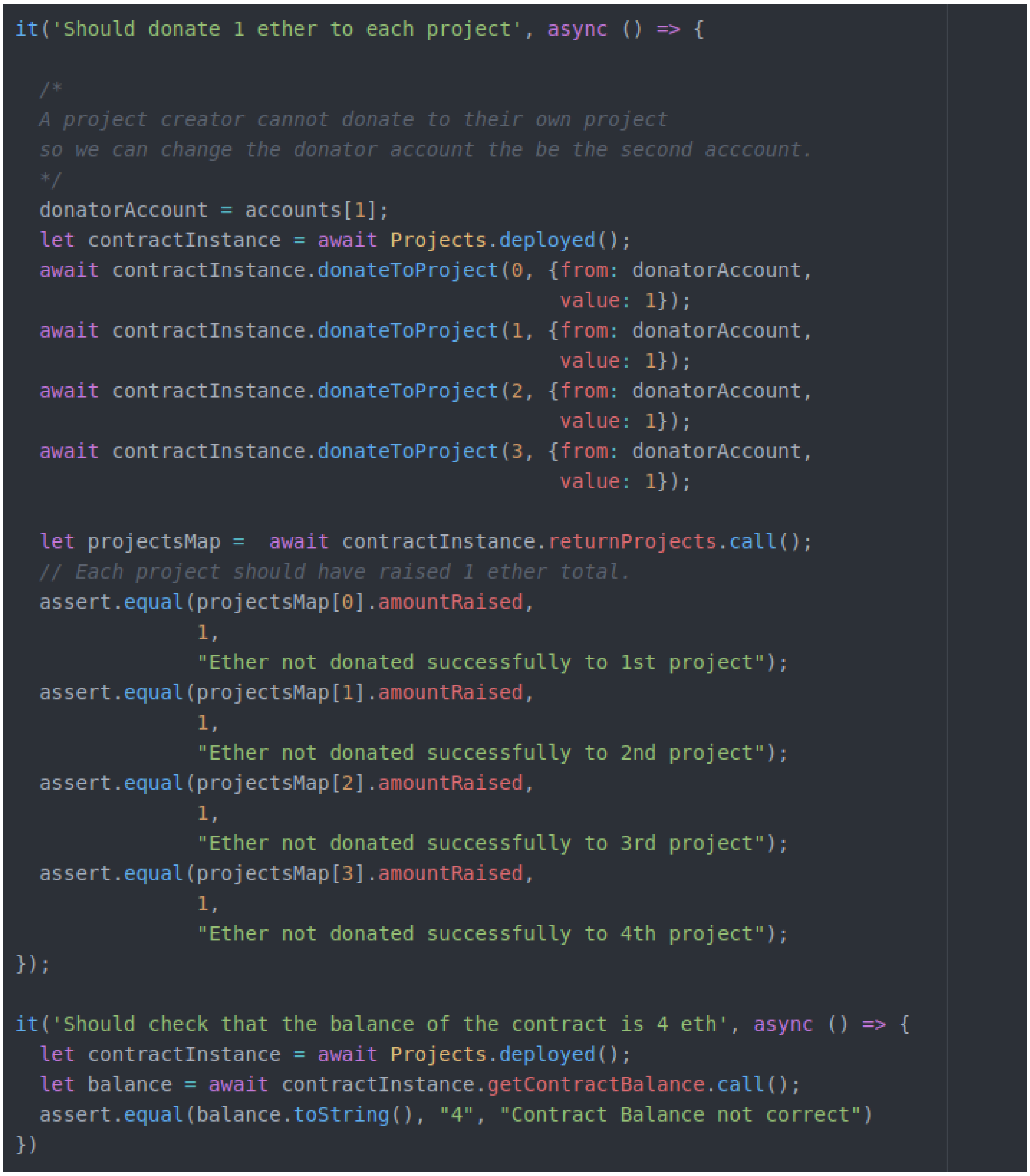
### 4.1.3. Testing return project's function



```
it('Should return 4 projects in the correct format', async () => {
  let contractInstance = await Projects.deployed();

  // the return projects function retruns an array of maps
  // the keys of the maps are the atrribute names which point to their value
  let projectsMap = await contractInstance.returnProjects.call();
  // should contain 4 projects
  assert.equal(projectsMap.length, 4, "Projects Map not the correct length");
  // and all the relevant keys
  assert.equal(("creatorAccount" in projectsMap[1]),
                true,
                "Project missing creator account key");
  assert.equal(("projectInfoHash" in projectsMap[1]),
                true,
                "Project missing projectInfoHash key");
  assert.equal(("fundingGoal" in projectsMap[1]),
                true,
                "Project missing funding goal key");
  assert.equal(("amountRaised" in projectsMap[1]),
                true,
                "Project missing amount raised key");
  assert.equal(("projectEndTime" in projectsMap[1]),
                true,
                "Project missing project end time key");
});

it('Should donate 1 ether to each project', async () => {
```

This test checks that the return project's function is returning the correct number of projects in the correct format. After running the tests in the previous subsection there should be 4 projects and they should contain the keys expected. We just check the keys for 1 of the projects here for efficiency as the projects are created in the same way, if one is correct the rest should be also.

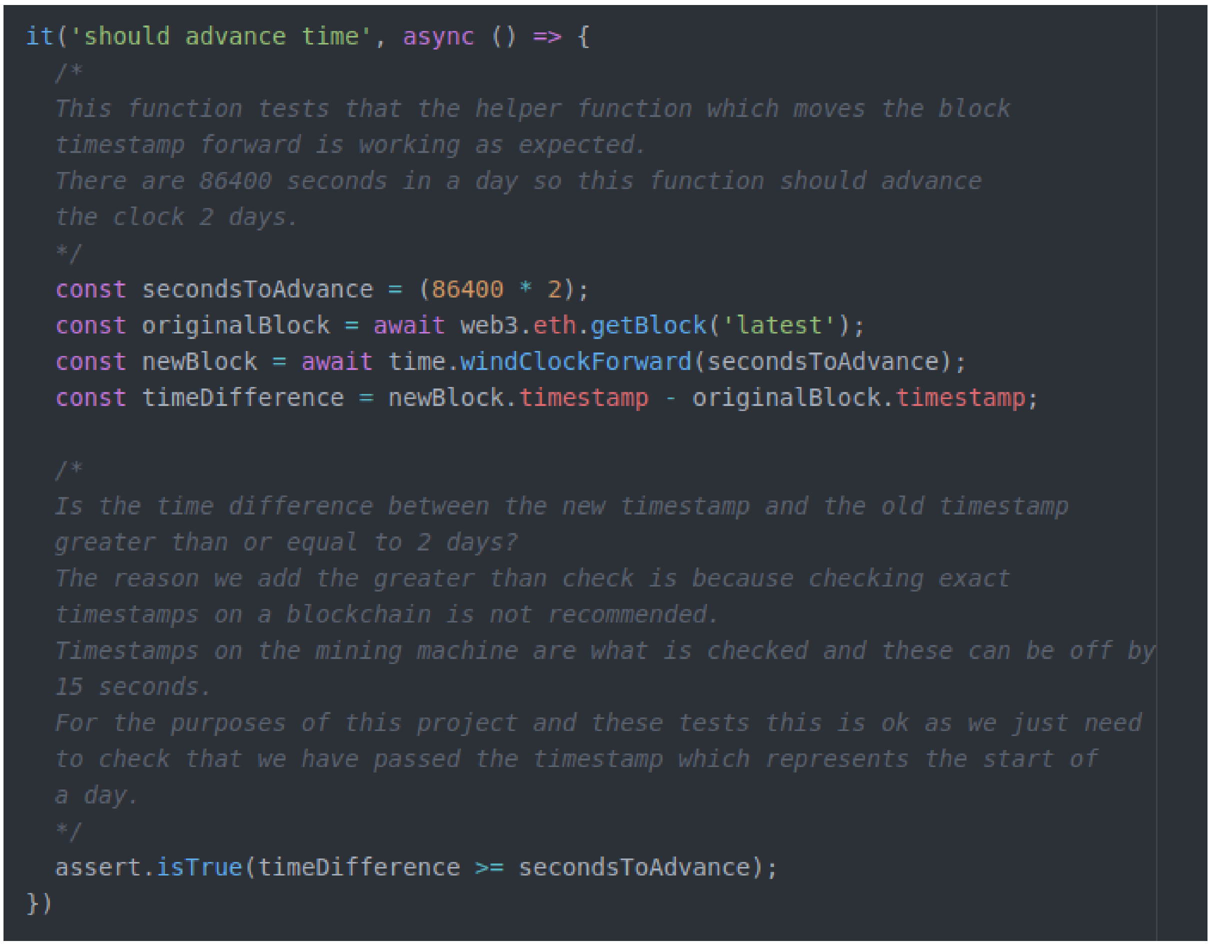
### 4.1.4. Testing donate to project function



```
it('Should donate 1 ether to each project', async () => {

  /*
  A project creator cannot donate to their own project
  so we can change the donator account the be the second acccount.
  */
  donatorAccount = accounts[1];
  let contractInstance = await Projects.deployed();
  await contractInstance.donateToProject(0, {from: donatorAccount,
                                              value: 1});
  await contractInstance.donateToProject(1, {from: donatorAccount,
                                              value: 1});
  await contractInstance.donateToProject(2, {from: donatorAccount,
                                              value: 1});
  await contractInstance.donateToProject(3, {from: donatorAccount,
                                              value: 1});

  let projectsMap =  await contractInstance.returnProjects.call();
  // Each project should have raised 1 ether total.
  assert.equal(projectsMap[0].amountRaised,
               1,
               "Ether not donated successfully to 1st project");
  assert.equal(projectsMap[1].amountRaised,
               1,
               "Ether not donated successfully to 2nd project");
  assert.equal(projectsMap[2].amountRaised,
               1,
               "Ether not donated successfully to 3rd project");
  assert.equal(projectsMap[3].amountRaised,
               1,
               "Ether not donated successfully to 4th project");
});

it('Should check that the balance of the contract is 4 eth', async () => {
  let contractInstance = await Projects.deployed();
  let balance = await contractInstance.getContractBalance.call();
  assert.equal(balance.toString(), "4", "Contract Balance not correct")
})
```

This test checks the donate to projects function. At the start we set the donator account to be different to the projects creator account as project creators cannot donate to their own projects. We then donate 1 ether to each of the 4 projects. After calling the return projects function, we check that the amount raised for each is 1 and then check that the overall balance of the contract is 4.

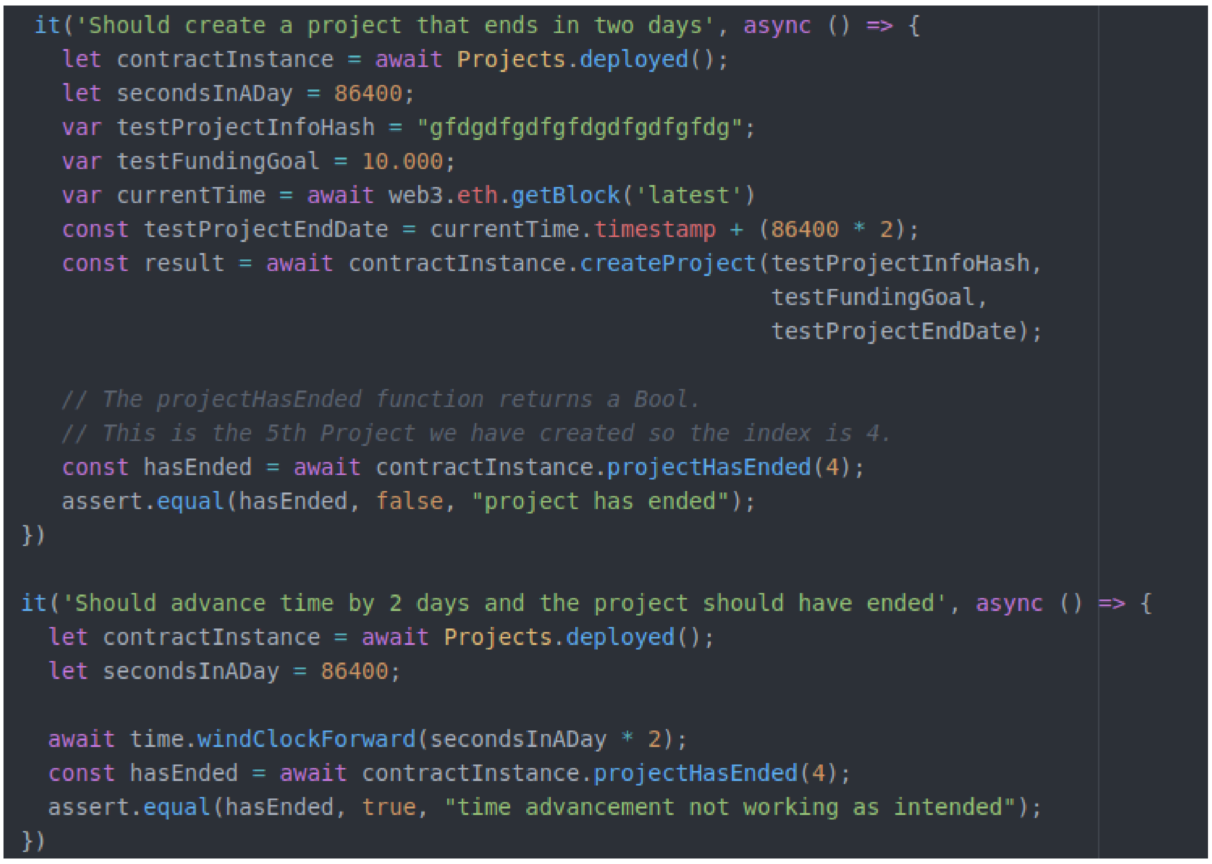
### 4.1.5. Testing the advance time helper function



```javascript
it('should advance time', async () => {
  /*
  This function tests that the helper function which moves the block
  timestamp forward is working as expected.
  There are 86400 seconds in a day so this function should advance
  the clock 2 days.
  */
  const secondsToAdvance = (86400 * 2);
  const originalBlock = await web3.eth.getBlock('latest');
  const newBlock = await time.windClockForward(secondsToAdvance);
  const timeDifference = newBlock.timestamp - originalBlock.timestamp;

  /*
  Is the time difference between the new timestamp and the old timestamp
  greater than or equal to 2 days?
  The reason we add the greater than check is because checking exact
  timestamps on a blockchain is not recommended.
  Timestamps on the mining machine are what is checked and these can be off by
  15 seconds.
  For the purposes of this project and these tests this is ok as we just need
  to check that we have passed the timestamp which represents the start of
  a day.
  */
  assert.isTrue(timeDifference >= secondsToAdvance);
})
```

This test ensures our helper function for advancing time is working. In order to test our pay refunds function we need to simulate a project reaching its end date. To do this we use built in methods of Ganache to mock time increase and block mining. This function notes the current timestamp, winds the clock forward and gets a new timestamp. It then checks that the new timestamp is greater than the old one meaning that time has gone forward and that a block has been mined.

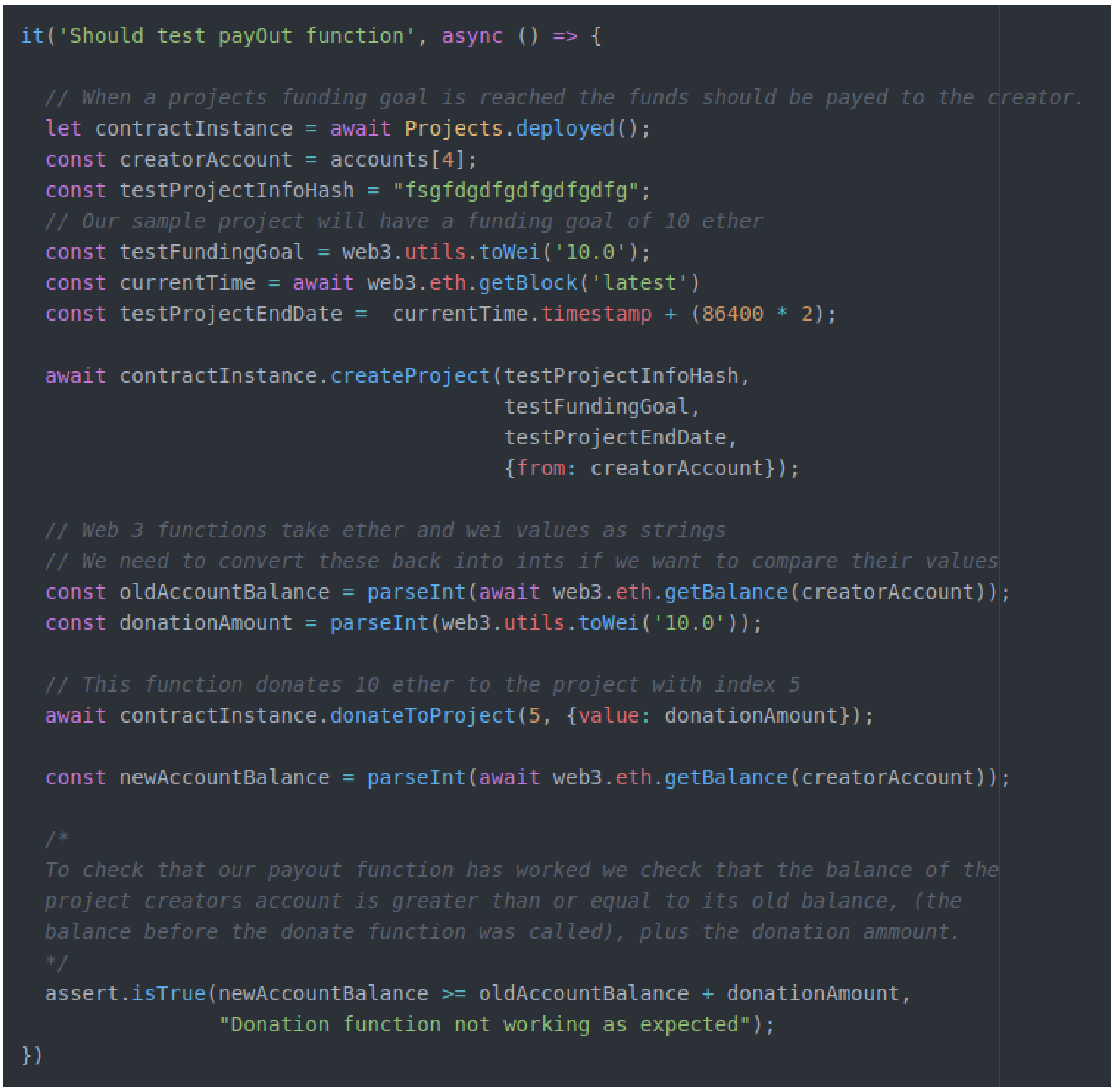
### 4.1.6. Testing mock project ending by date



```
 it('Should create a project that ends in two days', async () => {
   let contractInstance = await Projects.deployed();
   let secondsInADay = 86400;
   var testProjectInfoHash = "gfdgdfgdfgfdgdfgdfgfdg";
   var testFundingGoal = 10.000;
   var currentTime = await web3.eth.getBlock('latest')
   const testProjectEndDate = currentTime.timestamp + (86400 * 2);
   const result = await contractInstance.createProject(testProjectInfoHash,
                                                       testFundingGoal,
                                                       testProjectEndDate);

   // The projectHasEnded function returns a Bool.
   // This is the 5th Project we have created so the index is 4.
   const hasEnded = await contractInstance.projectHasEnded(4);
   assert.equal(hasEnded, false, "project has ended");
})

it('Should advance time by 2 days and the project should have ended', async () => {
  let contractInstance = await Projects.deployed();
  let secondsInADay = 86400;

  await time.windClockForward(secondsInADay * 2);
  const hasEnded = await contractInstance.projectHasEnded(4);
  assert.equal(hasEnded, true, "time advancement not working as intended");
})
```

This further tests our advance time function by creating a project that ends in 2 days, winding the clock forward by 2 days and checking that the project has now ended.

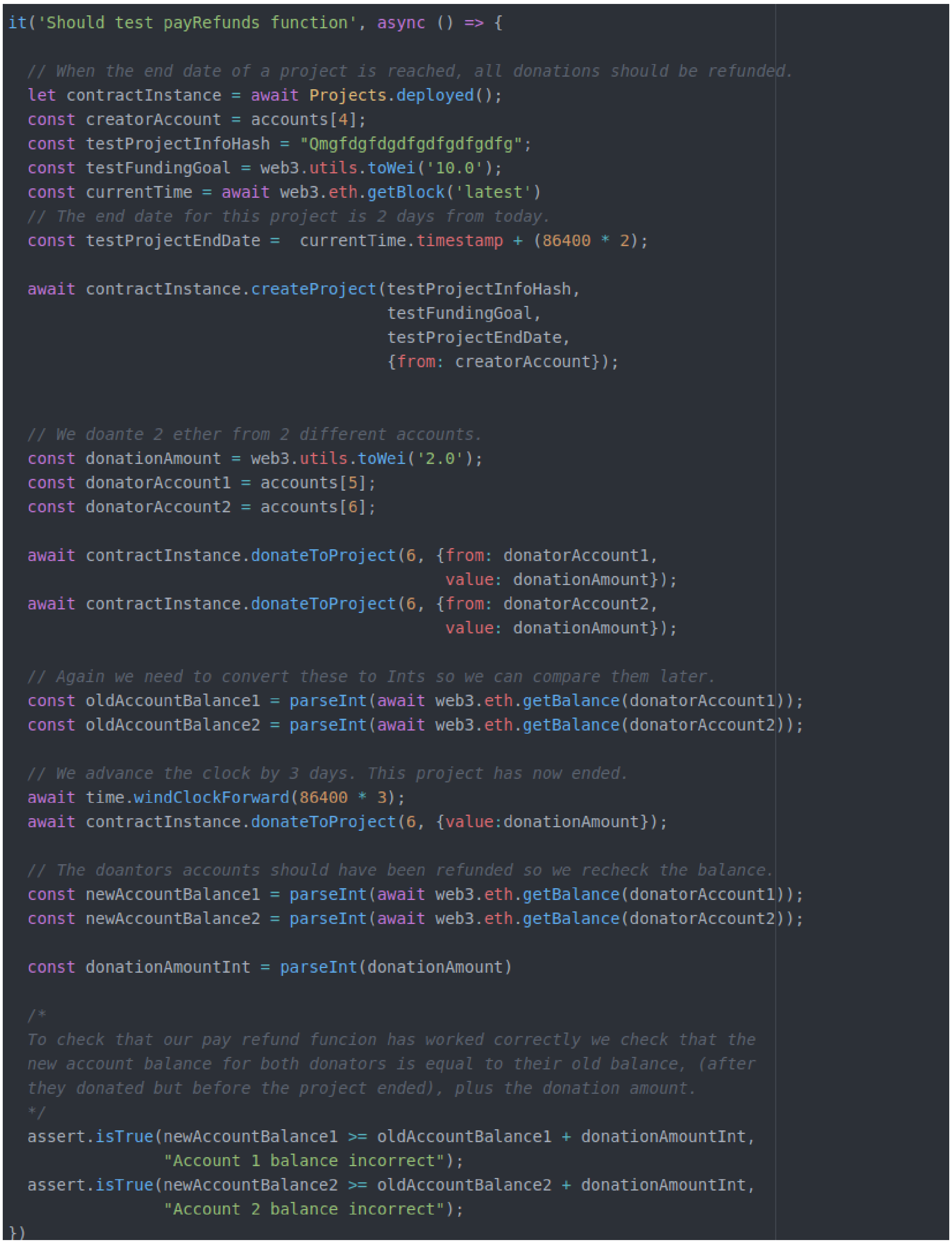
### 4.1.7. End-to-end pay out to project creator test



```javascript
it('Should test payOut function', async () => {

  // When a projects funding goal is reached the funds should be payed to the creator.
  let contractInstance = await Projects.deployed();
  const creatorAccount = accounts[4];
  const testProjectInfoHash = "fsgfdgdfgdfgdfgdfg";
  // Our sample project will have a funding goal of 10 ether
  const testFundingGoal = web3.utils.toWei('10.0');
  const currentTime = await web3.eth.getBlock('latest')
  const testProjectEndDate =  currentTime.timestamp + (86400 * 2);

  await contractInstance.createProject(testProjectInfoHash,
                                       testFundingGoal,
                                       testProjectEndDate,
                                       {from: creatorAccount});

  // Web 3 functions take ether and wei values as strings
  // We need to convert these back into ints if we want to compare their values
  const oldAccountBalance = parseInt(await web3.eth.getBalance(creatorAccount));
  const donationAmount = parseInt(web3.utils.toWei('10.0'));

  // This function donates 10 ether to the project with index 5
  await contractInstance.donateToProject(5, {value: donationAmount});

  const newAccountBalance = parseInt(await web3.eth.getBalance(creatorAccount));

  /*
  To check that our payout function has worked we check that the balance of the
  project creators account is greater than or equal to its old balance, (the
  balance before the donate function was called), plus the donation ammount.
  */
  assert.isTrue(newAccountBalance >= oldAccountBalance + donationAmount,
                "Donation function not working as expected");
})
```

This end-to-end test checks the payout function. First we create a project with account[4] from Ganache and set the funding goal to be 5 Eth. We then get the balance of account[4] and save it as a variable for checking later. We then donate 5 Eth from account[0], (the default account), to that newly created project, (it has index 5). This should end the project as the goal has been reached and therefore payout the amount raised to account[4]. We get the current balance of account[4] and check that is now equal to the old balance plus the donation amount.

### 4.1.8. End-to-end pay refunds test

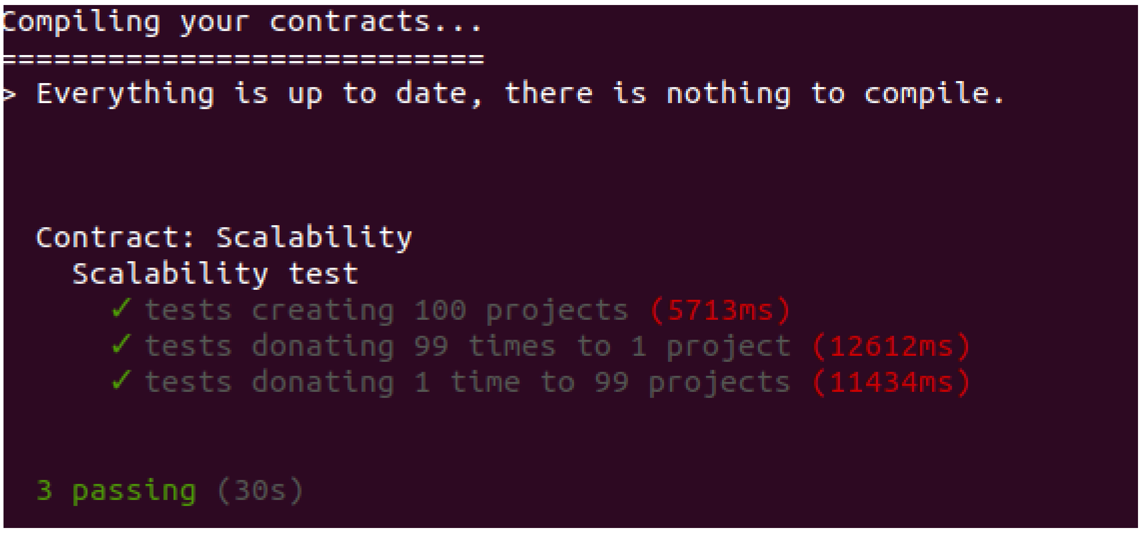


```
it('Should test payRefunds function', async () => {

  // When the end date of a project is reached, all donations should be refunded.
  let contractInstance = await Projects.deployed();
  const creatorAccount = accounts[4];
  const testProjectInfoHash = "Qmgfdgfdgdfgdfgdfgdfg";
  const testFundingGoal = web3.utils.toWei('10.0');
  const currentTime = await web3.eth.getBlock('latest')
  // The end date for this project is 2 days from today.
  const testProjectEndDate =  currentTime.timestamp + (86400 * 2);

  await contractInstance.createProject(testProjectInfoHash,
                                       testFundingGoal,
                                       testProjectEndDate,
                                       {from: creatorAccount});


  // We doante 2 ether from 2 different accounts.
  const donationAmount = web3.utils.toWei('2.0');
  const donatorAccount1 = accounts[5];
  const donatorAccount2 = accounts[6];

  await contractInstance.donateToProject(6, {from: donatorAccount1,
                                             value: donationAmount});
  await contractInstance.donateToProject(6, {from: donatorAccount2,
                                             value: donationAmount});

  // Again we need to convert these to Ints so we can compare them later.
  const oldAccountBalance1 = parseInt(await web3.eth.getBalance(donatorAccount1));
  const oldAccountBalance2 = parseInt(await web3.eth.getBalance(donatorAccount2));

  // We advance the clock by 3 days. This project has now ended.
  await time.windClockForward(86400 * 3);
  await contractInstance.donateToProject(6, {value:donationAmount});

  // The doantors accounts should have been refunded so we recheck the balance.
  const newAccountBalance1 = parseInt(await web3.eth.getBalance(donatorAccount1));
  const newAccountBalance2 = parseInt(await web3.eth.getBalance(donatorAccount2));

  const donationAmountInt = parseInt(donationAmount)

  /*
  To check that our pay refund funcion has worked correctly we check that the
  new account balance for both donators is equal to their old balance, (after
  they donated but before the project ended), plus the donation amount.
  */
  assert.isTrue(newAccountBalance1 >= oldAccountBalance1 + donationAmountInt,
                "Account 1 balance incorrect");
  assert.isTrue(newAccountBalance2 >= oldAccountBalance2 + donationAmountInt,
                "Account 2 balance incorrect");
})
```

This end-to-end function checks the pay refunds function. Fist we create a project that ends in 2 days with account[4]. We then donate 2 Ether to that project from each of account[5] and account[6] which will take 2 ether from them and hold it in the contract. We then record the balance of both accounts to be checked later. We then wind the clock forward by 2 days and the project should now be over. We make another donation which will check that the project has ended and fire the pay refunds function. We would now expect the funds to be returned to both account[5] and account[6]. We check the new balance and that it is equal to the old balance plus 2 Ether.

## 4.2. Smart contract scalability and gas usage testing

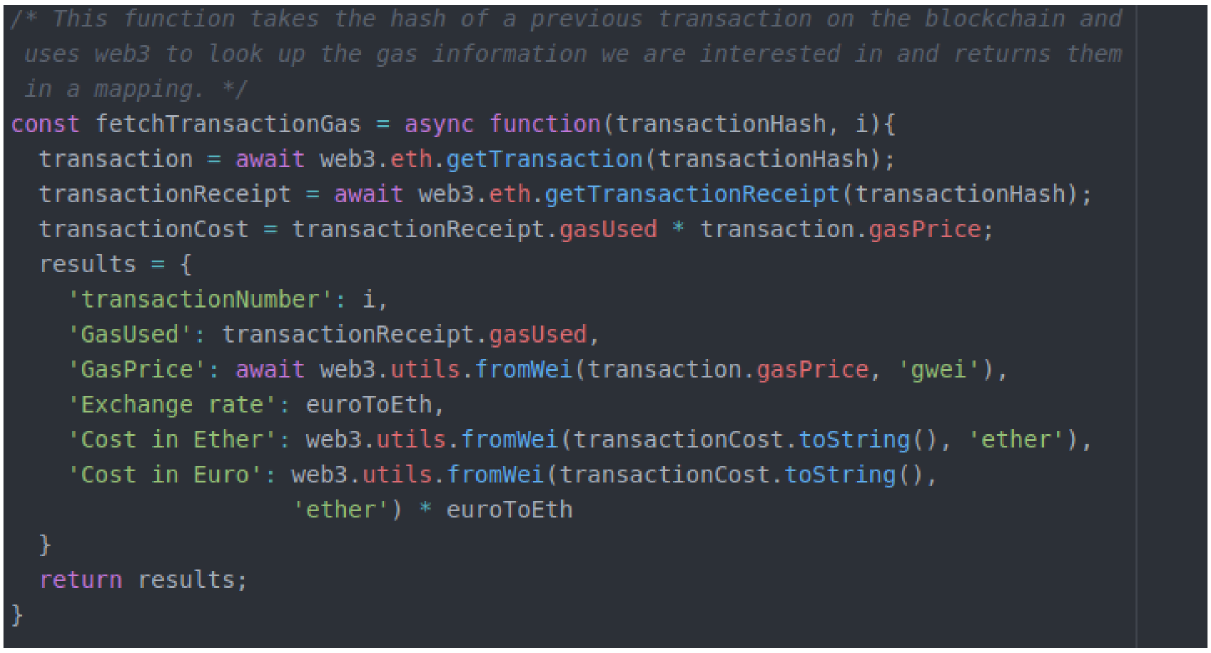
These tests are also written in JavaScript and run with truffles built in testing framework. The tests are in the scalability.js file in /src/test. To run the tests, you must have a local Ganache blockchain running on port 8546. For the purposes of these tests you must run the Ganache blockchain with 100 accounts rather than the default 10. Having more accounts means we can test the functions when many accounts are interacting with the smart contract. Results of the testing are written to log files in /src/test/logs. An example of the output when running the tests is seen below.



```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



  Contract: Scalability
    Scalability test
      ✓ tests creating 100 projects (5713ms)
      ✓ tests donating 99 times to 1 project (12612ms)
      ✓ tests donating 1 time to 99 projects (11434ms)


  3 passing (30s)
```

These tests are creating 100s of projects and calling 100s of functions so we would expect them to take a bit of time.

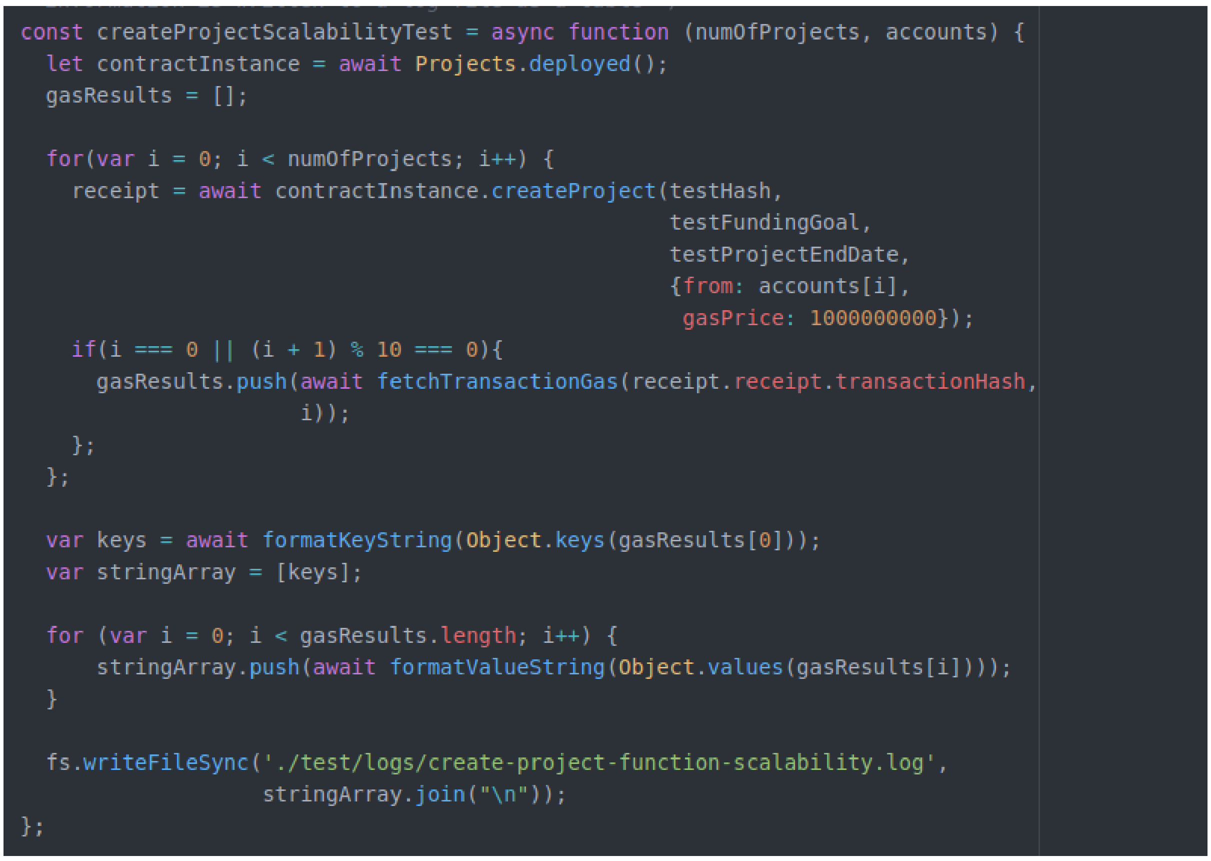
### 4.2.1. Building the maps for our log files



```
/* This function takes the hash of a previous transaction on the blockchain and
 uses web3 to look up the gas information we are interested in and returns them
 in a mapping. */
const fetchTransactionGas = async function(transactionHash, i){
  transaction = await web3.eth.getTransaction(transactionHash);
  transactionReceipt = await web3.eth.getTransactionReceipt(transactionHash);
  transactionCost = transactionReceipt.gasUsed * transaction.gasPrice;
  results = {
    'transactionNumber': i,
    'GasUsed': transactionReceipt.gasUsed,
    'GasPrice': await web3.utils.fromWei(transaction.gasPrice, 'gwei'),
    'Exchange rate': euroToEth,
    'Cost in Ether': web3.utils.fromWei(transactionCost.toString(), 'ether'),
    'Cost in Euro': web3.utils.fromWei(transactionCost.toString(),
                'ether') * euroToEth
  }
  return results;
}
```

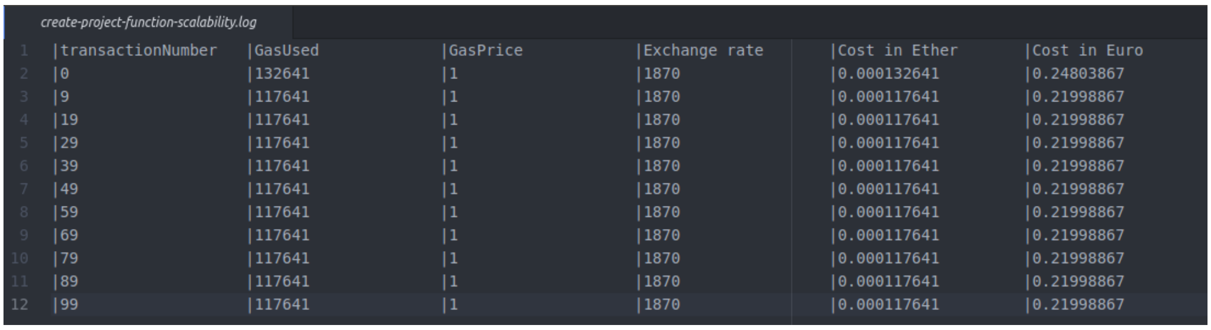
This function takes the hash of a previous transaction and returns a map with statistics that can tell us about the gas usage.

### 4.2.2. Creating projects function



```
const createProjectScalabilityTest = async function (numOfProjects, accounts) {
  let contractInstance = await Projects.deployed();
  gasResults = [];

  for(var i = 0; i < numOfProjects; i++) {
    receipt = await contractInstance.createProject(testHash,
                                                   testFundingGoal,
                                                   testProjectEndDate,
                                                   {from: accounts[i],
                                                    gasPrice: 1000000000});
    if(i === 0 || (i + 1) % 10 === 0){
      gasResults.push(await fetchTransactionGas(receipt.receipt.transactionHash,
                      i));
    };
  };

  var keys = await formatKeyString(Object.keys(gasResults[0]));
  var stringArray = [keys];

  for (var i = 0; i < gasResults.length; i++) {
      stringArray.push(await formatValueString(Object.values(gasResults[i])));
  }

  fs.writeFileSync('./test/logs/create-project-function-scalability.log',
                   stringArray.join("\n"));
};
```

This function takes a number and creates that number of projects. For the first transaction and then for every transaction that is a multiple of ten it records the gas usage information. It then takes that information and pushes it to a string array before writing t to a table in a log file.

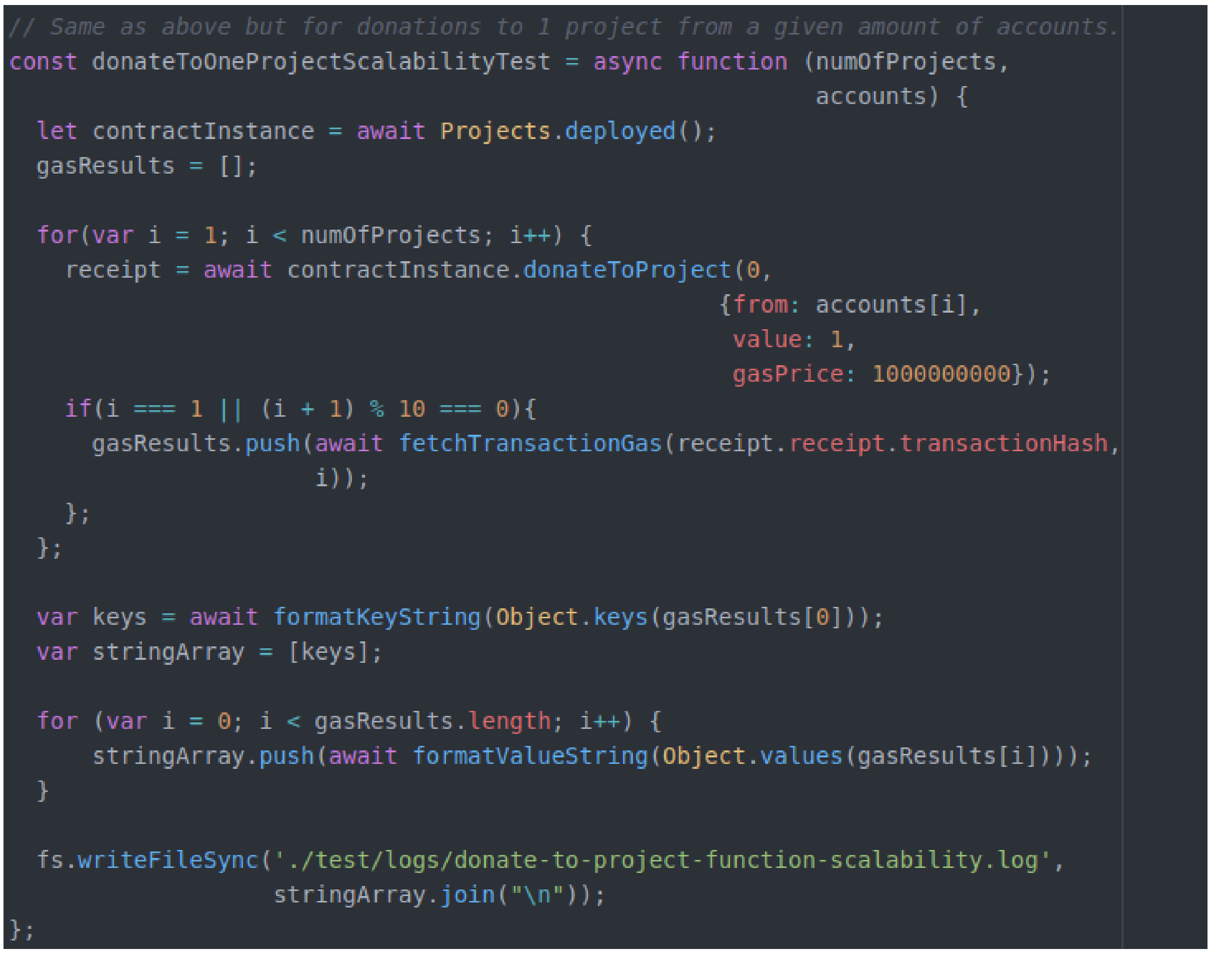


create-project-function-scalability.log

```
|transactionNumber  |GasUsed        |GasPrice       |Exchange rate  |Cost in Ether  |Cost in Euro
|0                  |132641         |1              |1870           |0.000132641    |0.24803867
|9                  |117641         |1              |1870           |0.000117641    |0.21998867
|19                 |117641         |1              |1870           |0.000117641    |0.21998867
|29                 |117641         |1              |1870           |0.000117641    |0.21998867
|39                 |117641         |1              |1870           |0.000117641    |0.21998867
|49                 |117641         |1              |1870           |0.000117641    |0.21998867
|59                 |117641         |1              |1870           |0.000117641    |0.21998867
|69                 |117641         |1              |1870           |0.000117641    |0.21998867
|79                 |117641         |1              |1870           |0.000117641    |0.21998867
|89                 |117641         |1              |1870           |0.000117641    |0.21998867
|99                 |117641         |1              |1870           |0.000117641    |0.21998867
```

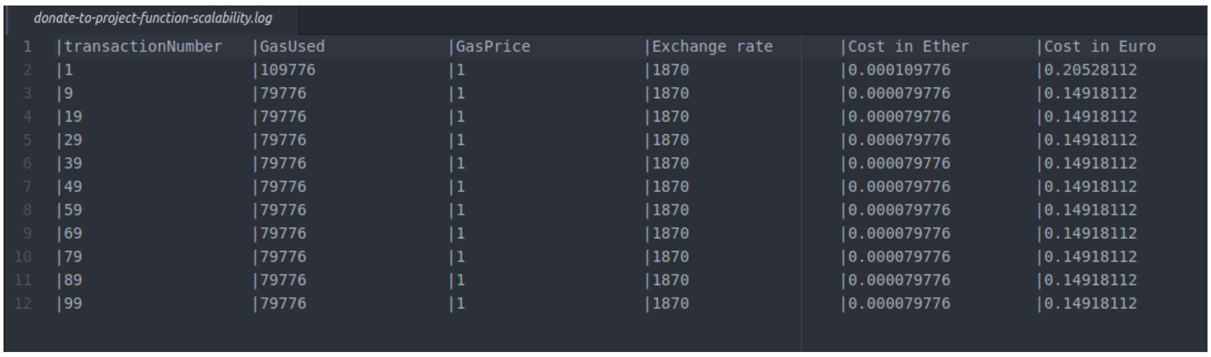
Here is an example of the log file output when this is run 100 times. As you can see the gas usage is not increasing with the number of calls. This means the gas usage is linear and therefore the function is scalable.

## 4.2.3. Donate to single project multiple times



```javascript
// Same as above but for donations to 1 project from a given amount of accounts.
const donateToOneProjectScalabilityTest = async function (numOfProjects,
                                                           accounts) {
  let contractInstance = await Projects.deployed();
  gasResults = [];

  for(var i = 1; i < numOfProjects; i++) {
    receipt = await contractInstance.donateToProject(0,
                                                    {from: accounts[i],
                                                     value: 1,
                                                     gasPrice: 1000000000});

    if(i === 1 || (i + 1) % 10 === 0){
      gasResults.push(await fetchTransactionGas(receipt.receipt.transactionHash,
                      i));
    };
  };

  var keys = await formatKeyString(Object.keys(gasResults[0]));
  var stringArray = [keys];

  for (var i = 0; i < gasResults.length; i++) {
      stringArray.push(await formatValueString(Object.values(gasResults[i])));
  }

  fs.writeFileSync('./test/logs/donate-to-project-function-scalability.log',
                   stringArray.join("\n"));
};
```

This function takes a number and donates to one project that number of times. For the first transaction and then for every transaction that is a multiple of ten it records the gas usage information. It then takes that information and pushes it to a string array before writing t to a table in a log file.

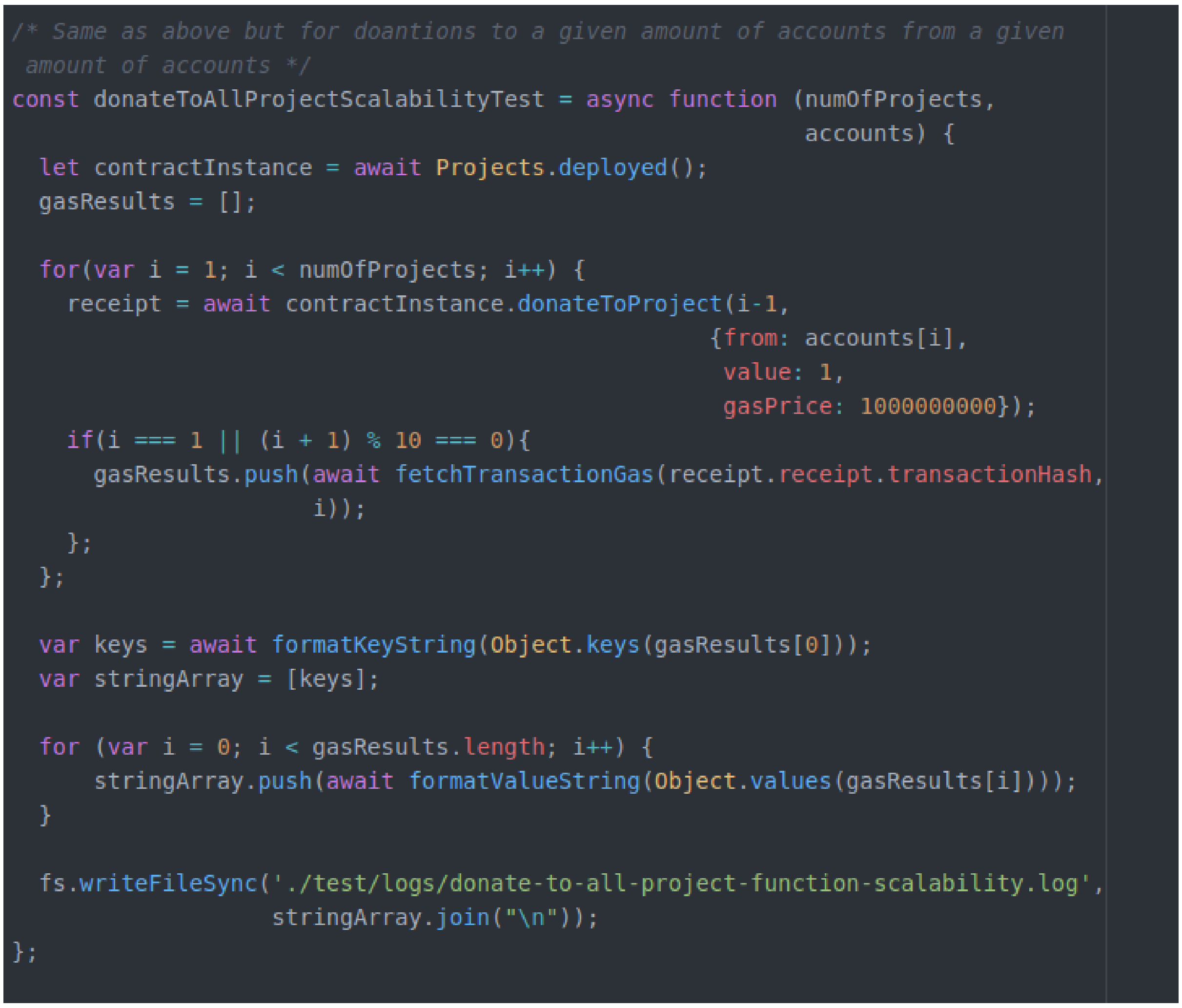


donate-to-project-function-scalability.log

```
|transactionNumber   |GasUsed             |GasPrice            |Exchange rate       |Cost in Ether       |Cost in Euro
|1                   |109776              |1                   |1870                |0.000109776         |0.20528112
|9                   |79776               |1                   |1870                |0.000079776         |0.14918112
|19                  |79776               |1                   |1870                |0.000079776         |0.14918112
|29                  |79776               |1                   |1870                |0.000079776         |0.14918112
|39                  |79776               |1                   |1870                |0.000079776         |0.14918112
|49                  |79776               |1                   |1870                |0.000079776         |0.14918112
|59                  |79776               |1                   |1870                |0.000079776         |0.14918112
|69                  |79776               |1                   |1870                |0.000079776         |0.14918112
|79                  |79776               |1                   |1870                |0.000079776         |0.14918112
|89                  |79776               |1                   |1870                |0.000079776         |0.14918112
|99                  |79776               |1                   |1870                |0.000079776         |0.14918112
```

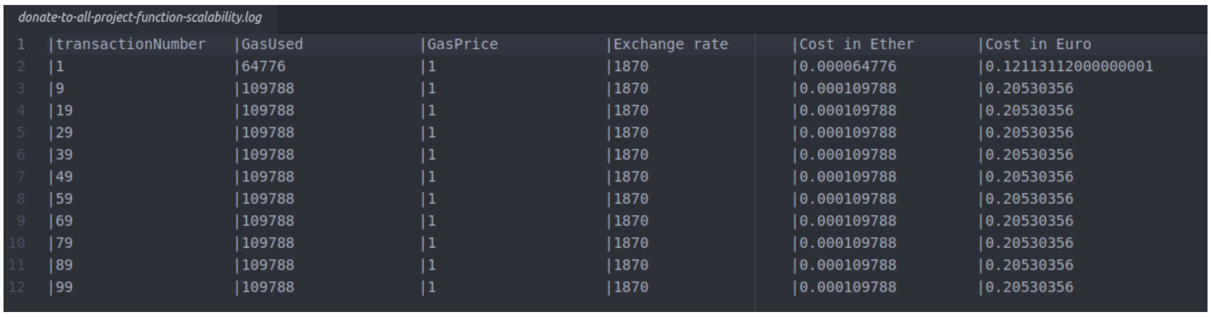
Here is an example of the log file output when this is run 100 times. As you can see the gas usage is not increasing with the number of calls. This means the gas usage is linear and therefore the function is scalable.

### 4.2.4. Donate to multiple projects



```
/* Same as above but for doantions to a given amount of accounts from a given
 amount of accounts */
const donateToAllProjectScalabilityTest = async function (numOfProjects,
                                                          accounts) {
  let contractInstance = await Projects.deployed();
  gasResults = [];

  for(var i = 1; i < numOfProjects; i++) {
    receipt = await contractInstance.donateToProject(i-1,
                                                   {from: accounts[i],
                                                    value: 1,
                                                    gasPrice: 1000000000});
    if(i === 1 || (i + 1) % 10 === 0){
      gasResults.push(await fetchTransactionGas(receipt.receipt.transactionHash,
                      i));
    };
  };

  var keys = await formatKeyString(Object.keys(gasResults[0]));
  var stringArray = [keys];

  for (var i = 0; i < gasResults.length; i++) {
      stringArray.push(await formatValueString(Object.values(gasResults[i])));
  }

  fs.writeFileSync('./test/logs/donate-to-all-project-function-scalability.log',
                   stringArray.join("\n"));
};
```

This function takes a number and donates to that number of projects from a different account every time. For the first transaction and then for every 10 transactions it records the gas usage information. It then takes that information and pushes it to a string array before writing t to a table in a log file.



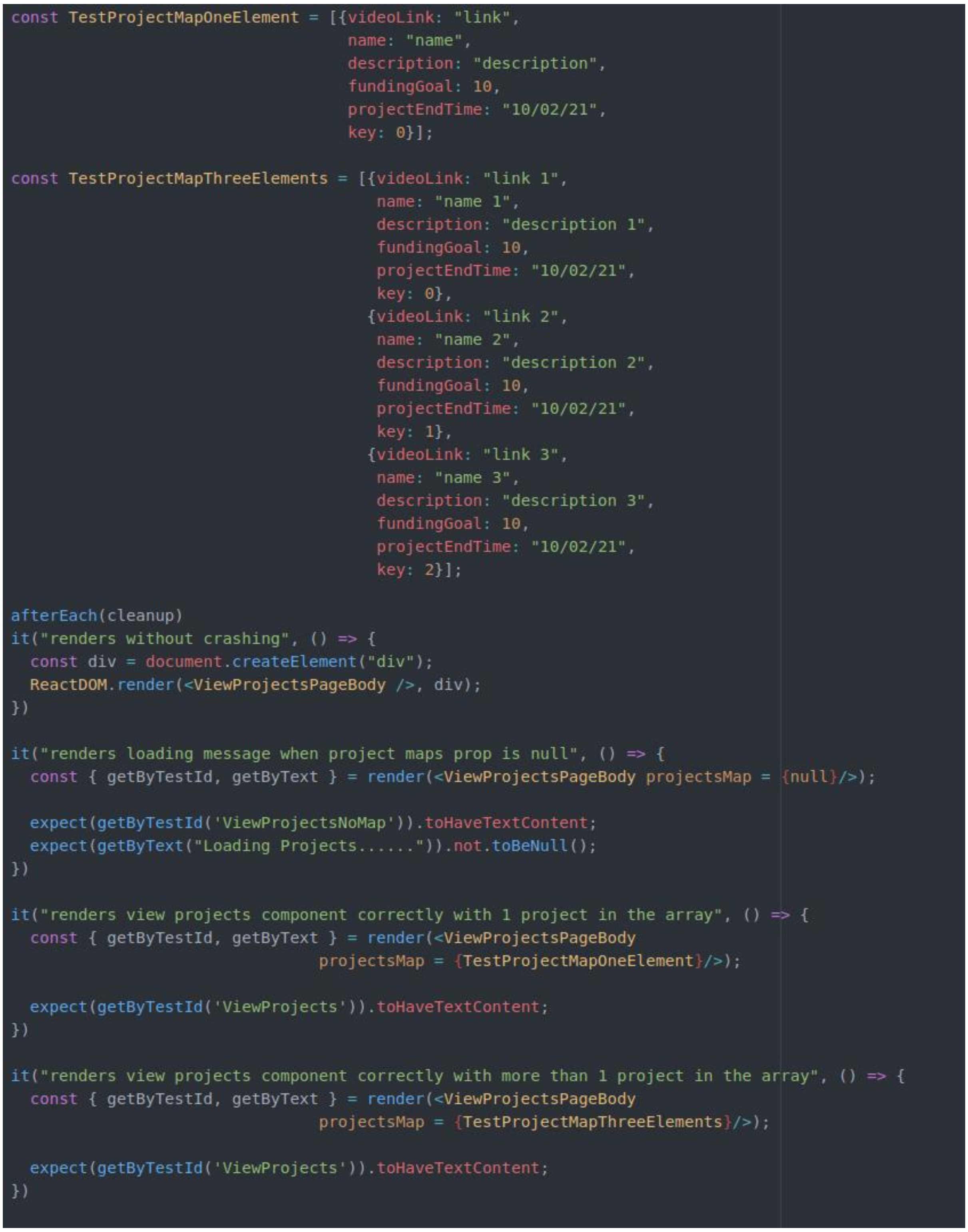
donate-to-all-project-function-scalability.log

```
|transactionNumber   |GasUsed              |GasPrice             |Exchange rate           |Cost in Ether        |Cost in Euro
|1                   |64776                |1                    |1870                    |0.000064776          |0.12113112000000001
|9                   |109788               |1                    |1870                    |0.000109788          |0.20530356
|19                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|29                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|39                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|49                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|59                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|69                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|79                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|89                  |109788               |1                    |1870                    |0.000109788          |0.20530356
|99                  |109788               |1                    |1870                    |0.000109788          |0.20530356
```

Here is an example of the log file output when this is run 99 times. As you can see the gas usage is not increasing with the number of calls. This means the gas usage is linear and therefore the function is scalable.

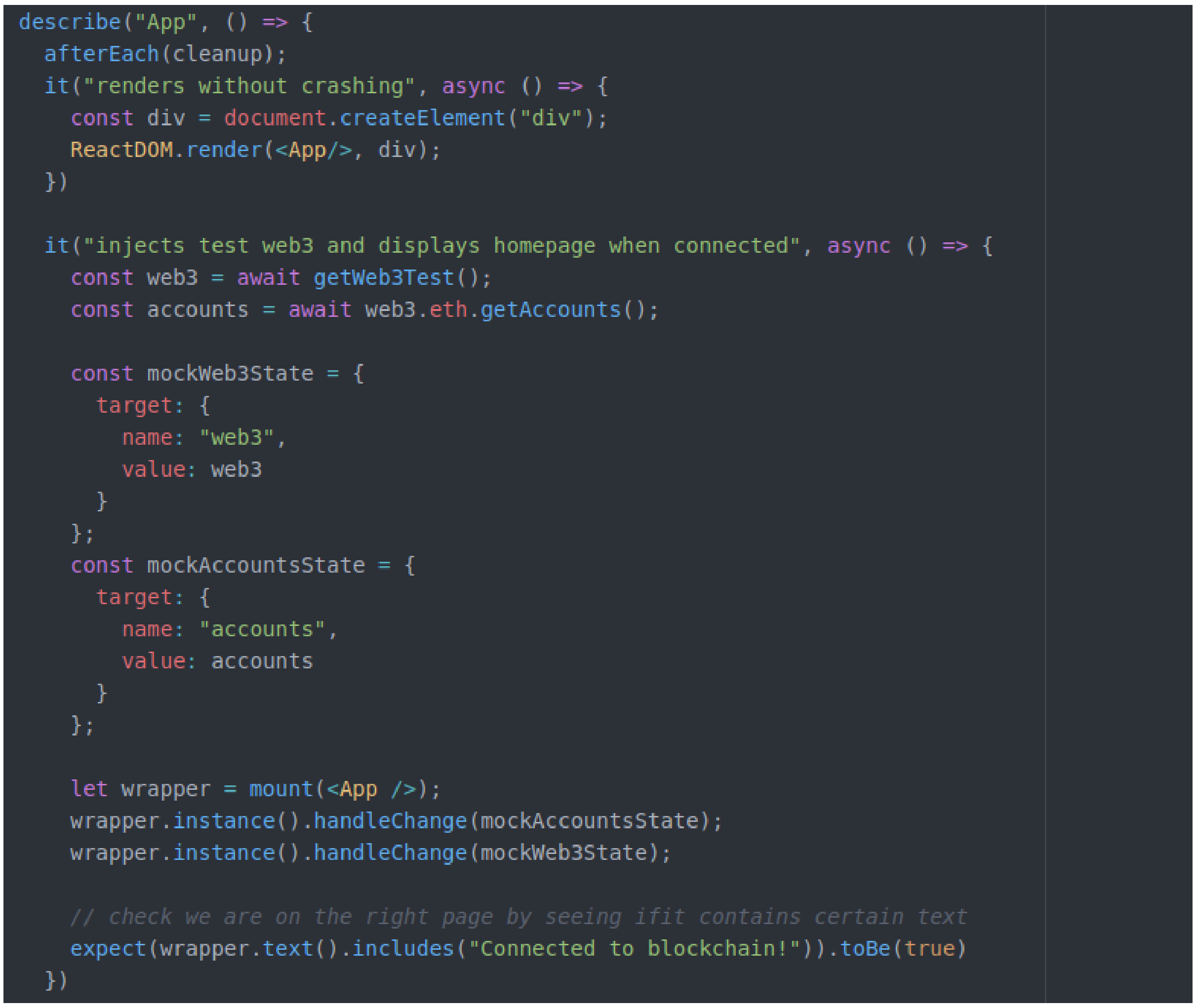
## 4.3. User interface testing

The user interface also has many tests. Each component has its own test to ensure it is rendering correctly. I will not list them all here as it would be too long but below is one example.



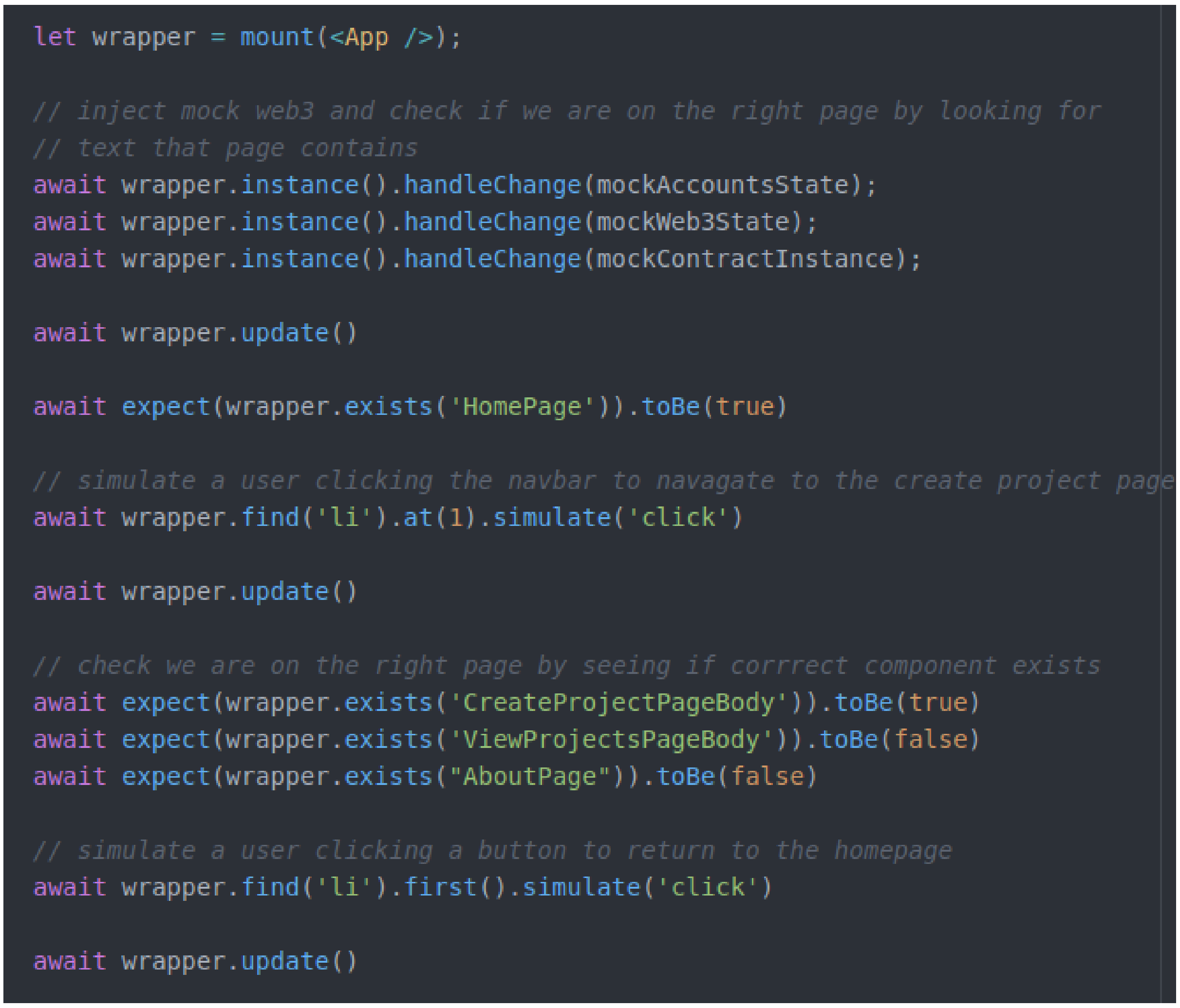
```
const TestProjectMapOneElement = [{videoLink: "link",
                                   name: "name",
                                   description: "description",
                                   fundingGoal: 10,
                                   projectEndTime: "10/02/21",
                                   key: 0}];

const TestProjectMapThreeElements = [{videoLink: "link 1",
                                      name: "name 1",
                                      description: "description 1",
                                      fundingGoal: 10,
                                      projectEndTime: "10/02/21",
                                      key: 0},
                                     {videoLink: "link 2",
                                      name: "name 2",
                                      description: "description 2",
                                      fundingGoal: 10,
                                      projectEndTime: "10/02/21",
                                      key: 1},
                                     {videoLink: "link 3",
                                      name: "name 3",
                                      description: "description 3",
                                      fundingGoal: 10,
                                      projectEndTime: "10/02/21",
                                      key: 2}];

afterEach(cleanup)
it("renders without crashing", () => {
  const div = document.createElement("div");
  ReactDOM.render(<ViewProjectsPageBody />, div);
})

it("renders loading message when project maps prop is null", () => {
  const { getByTestId, getByText } = render(<ViewProjectsPageBody projectsMap = {null}/>);

  expect(getByTestId('ViewProjectsNoMap')).toHaveTextContent;
  expect(getByText("Loading Projects......")).not.toBeNull();
})

it("renders view projects component correctly with 1 project in the array", () => {
  const { getByTestId, getByText } = render(<ViewProjectsPageBody
                                 projectsMap = {TestProjectMapOneElement}/>);

  expect(getByTestId('ViewProjects')).toHaveTextContent;
})

it("renders view projects component correctly with more than 1 project in the array", () => {
  const { getByTestId, getByText } = render(<ViewProjectsPageBody
                                 projectsMap = {TestProjectMapThreeElements}/>);

  expect(getByTestId('ViewProjects')).toHaveTextContent;
})
```

This test is for the view projects page. It ensures that the expected content is rendered in the case of the projects map still being loaded, the projects map containing a single project or the projects map containing multiple projects. We mock the various possible projects maps being in state with the test variables at the top of the page.



```
describe("App", () => {
  afterEach(cleanup);
  it("renders without crashing", async () => {
    const div = document.createElement("div");
    ReactDOM.render(<App/>, div);
  })

  it("injects test web3 and displays homepage when connected", async () => {
    const web3 = await getWeb3Test();
    const accounts = await web3.eth.getAccounts();

    const mockWeb3State = {
      target: {
        name: "web3",
        value: web3
      }
    };
    const mockAccountsState = {
      target: {
        name: "accounts",
        value: accounts
      }
    };

    let wrapper = mount(<App />);
    wrapper.instance().handleChange(mockAccountsState);
    wrapper.instance().handleChange(mockWeb3State);

    // check we are on the right page by seeing ifit contains certain text
    expect(wrapper.text().includes("Connected to blockchain!")).toBe(true)
  })
```

This test mocks web3 to ensure that the correct content is displayed after connecting to Metamask.

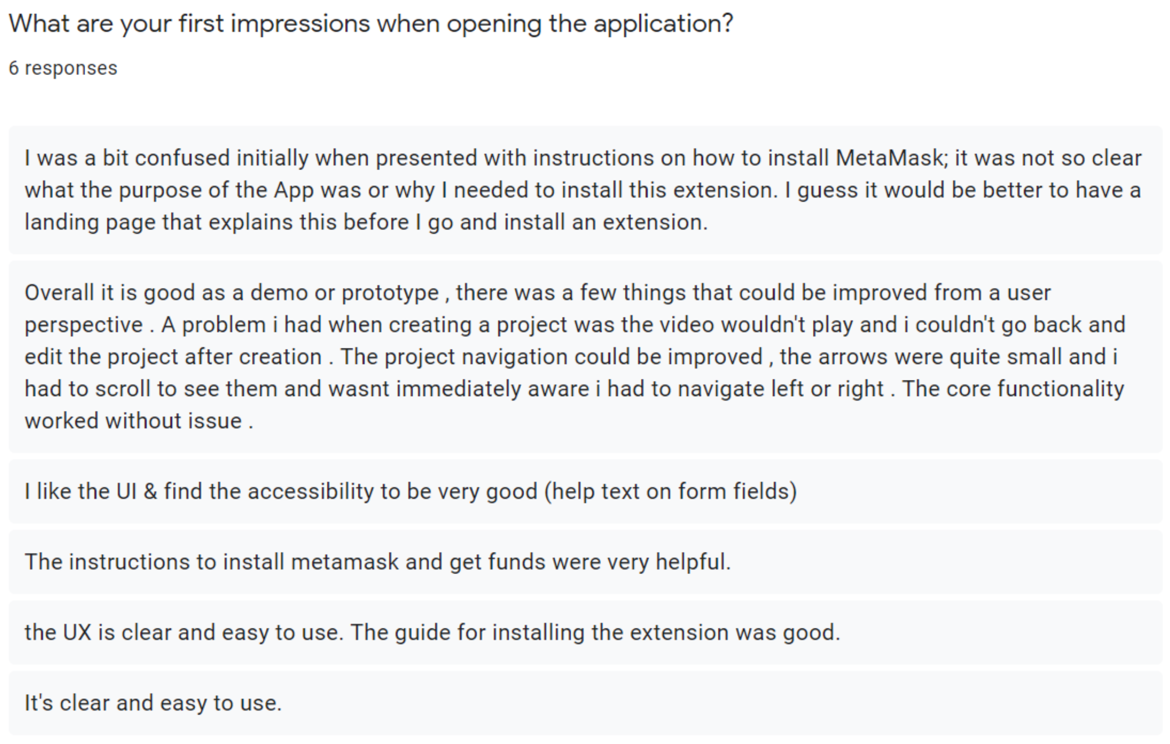


```
let wrapper = mount(<App />);

// inject mock web3 and check if we are on the right page by looking for
// text that page contains
await wrapper.instance().handleChange(mockAccountsState);
await wrapper.instance().handleChange(mockWeb3State);
await wrapper.instance().handleChange(mockContractInstance);

await wrapper.update()

await expect(wrapper.exists('HomePage')).toBe(true)

// simulate a user clicking the navbar to navagate to the create project page
await wrapper.find('li').at(1).simulate('click')

await wrapper.update()

// check we are on the right page by seeing if corrrect component exists
await expect(wrapper.exists('CreateProjectPageBody')).toBe(true)
await expect(wrapper.exists('ViewProjectsPageBody')).toBe(false)
await expect(wrapper.exists("AboutPage")).toBe(false)

// simulate a user clicking a button to return to the homepage
await wrapper.find('li').first().simulate('click')

await wrapper.update()
```

This is an excerpt of a long test which checks that navigation between the pages is working correctly. Here we mock web3 and connect to the app. We check that we are on the correct page by checking if the Homepage wrapper exists. We then find the second list item on the page and simulate a click on it. This list item is the navbar menu button for the create projects page. We then check we are on the correct page by checking that the Create Project Page wrapper exists and that no other age wrappers exist. The test carries on like this to ensure that all the navigation buttons are working correctly.

## 4.4. Usability testing and user feedback

For usability testing and to get user feedback for potential additions and changes to the UI a study was conducted. In involved having a user interact with a live version of the UI and completing a survey about their experience. The survey responses were anonymous. I have included some of the responses bellow.



What are your first impressions when opening the application?

6 responses

I was a bit confused initially when presented with instructions on how to install MetaMask; it was not so clear what the purpose of the App was or why I needed to install this extension. I guess it would be better to have a landing page that explains this before I go and install an extension.

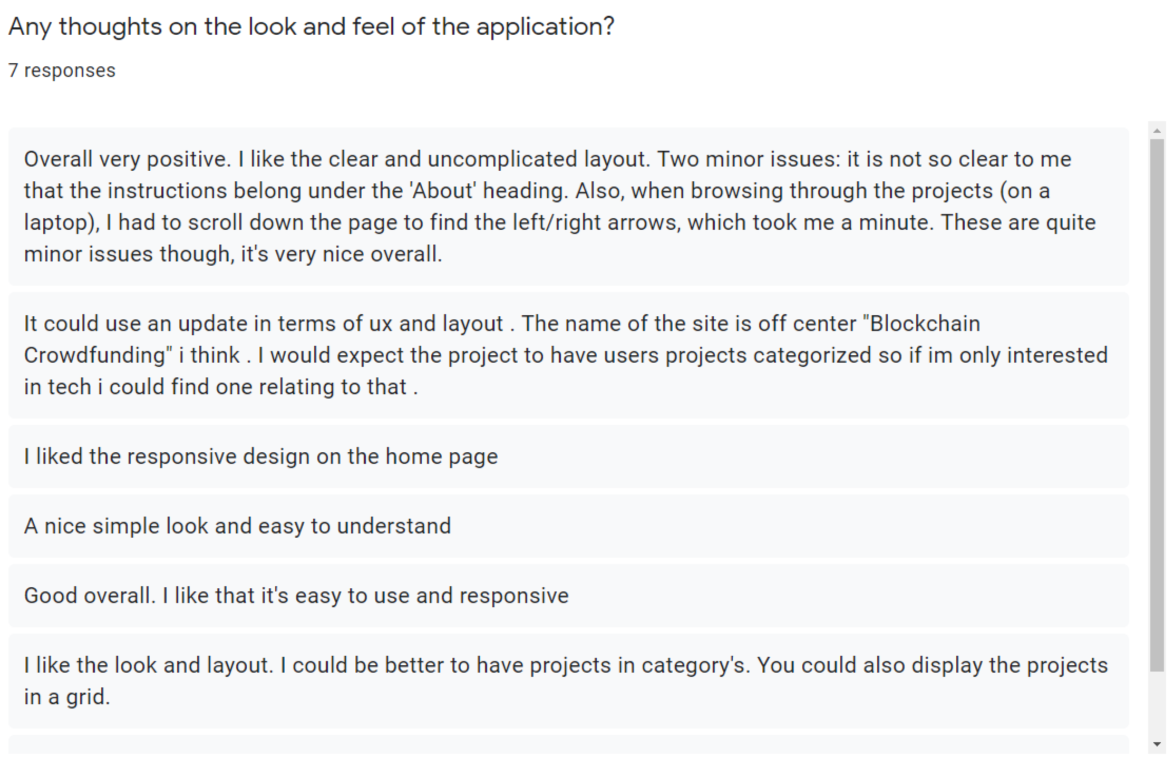
Overall it is good as a demo or prototype , there was a few things that could be improved from a user perspective . A problem i had when creating a project was the video wouldn't play and i couldn't go back and edit the project after creation . The project navigation could be improved , the arrows were quite small and i had to scroll to see them and wasnt immediately aware i had to navigate left or right . The core functionality worked without issue .

I like the UI & find the accessibility to be very good (help text on form fields)

The instructions to install metamask and get funds were very helpful.

the UX is clear and easy to use. The guide for installing the extension was good.

It's clear and easy to use.



Any thoughts on the look and feel of the application?

7 responses

Overall very positive. I like the clear and uncomplicated layout. Two minor issues: it is not so clear to me that the instructions belong under the 'About' heading. Also, when browsing through the projects (on a laptop), I had to scroll down the page to find the left/right arrows, which took me a minute. These are quite minor issues though, it's very nice overall.

It could use an update in terms of ux and layout . The name of the site is off center "Blockchain Crowdfunding" i think . I would expect the project to have users projects categorized so if im only interested in tech i could find one relating to that .

I liked the responsive design on the home page

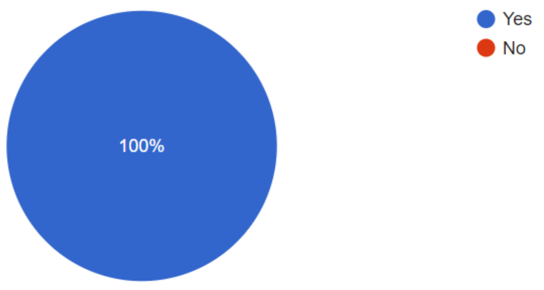
A nice simple look and easy to understand

Good overall. I like that it's easy to use and responsive

I like the look and layout. I could be better to have projects in category's. You could also display the projects in a grid.

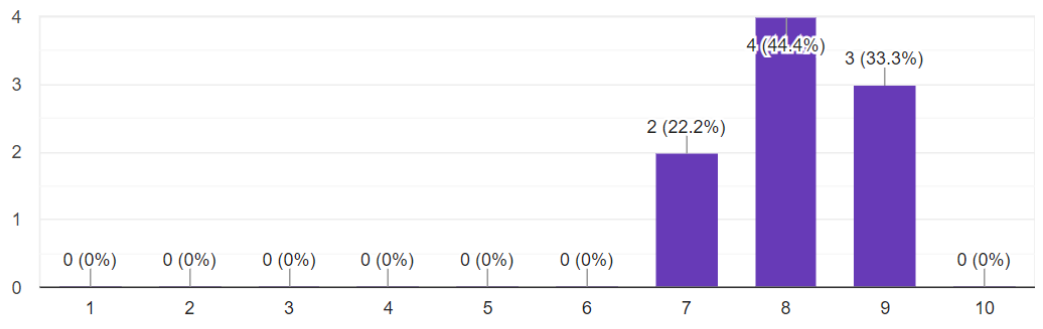
Is the purpose of this application clear to you?

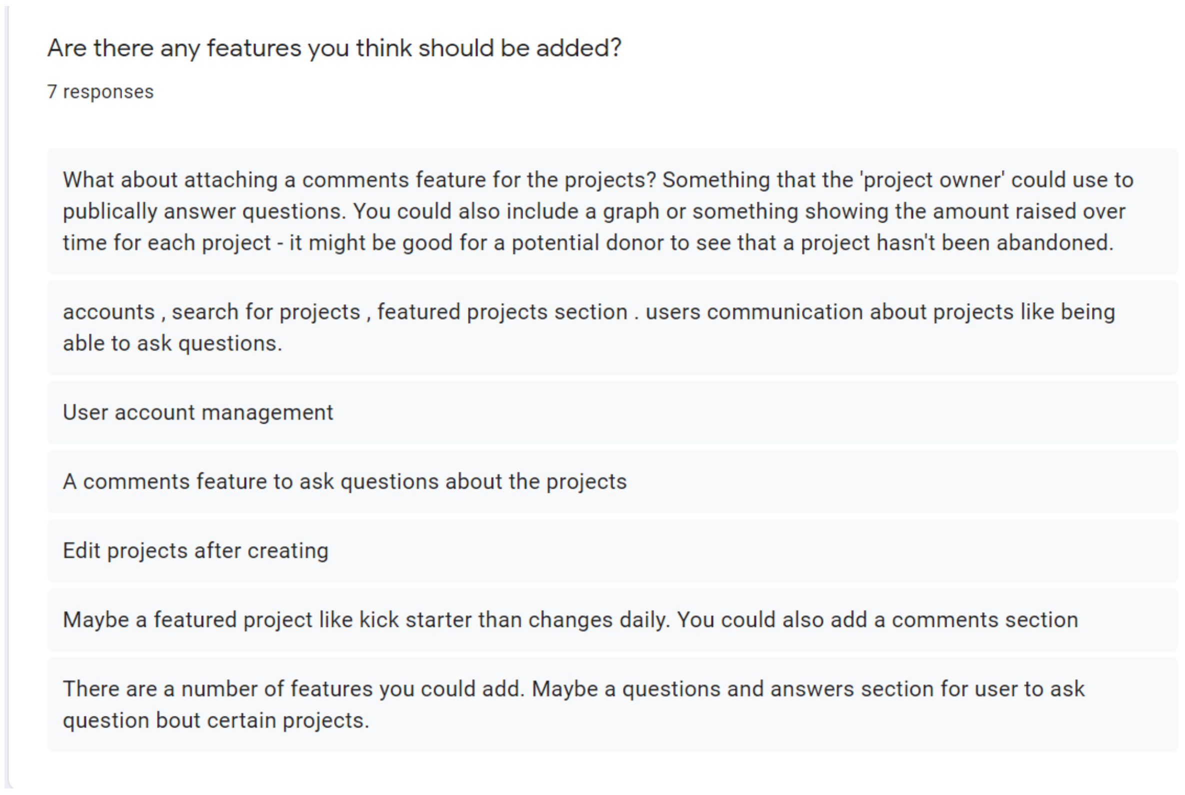
9 responses



How would you rate the look and feel of the application out of 10?

9 responses




Are there any features you think should be added?

7 responses

What about attaching a comments feature for the projects? Something that the 'project owner' could use to publically answer questions. You could also include a graph or something showing the amount raised over time for each project - it might be good for a potential donor to see that a project hasn't been abandoned.

accounts , search for projects , featured projects section . users communication about projects like being able to ask questions.

User account management

A comments feature to ask questions about the projects

Edit projects after creating

Maybe a featured project like kick starter than changes daily. You could also add a comments section

There are a number of features you could add. Maybe a questions and answers section for user to ask question bout certain projects.

Using this feedback, I was able to identify some improvements to make to the UI. Some users mentioned that the about page was misnamed as it contained instructions, so I renamed the page to help which I think better describes its content. Some users mentioned that the landing page did not feature an explanation of the app, just how to download and install Metamask. I added a component to the landing page with information as they suggested. Some users complained that the navigation arrows for navigating between projects were too far down the page so changed the placement. One user identified a bug when creating a project which led me to do more testing to get rid of the bug.

The users also had many suggestions for features that were out of the scope of this project but good ideas for future work. This includes things like a Q & A section to ask questions about the projects. Some users also wanted there to be a way to edit projects after creating them. This would be a nice feature but hard to implement due to the immutable nature of both the blockchain and IPFS. This could also be something to look at for future work.

# 5. Conclusion

## 5.1. Problems solved

- This project solves some of the issues with crowdfunding on traditional platforms. It removes the need for a middleman to transfer funds and the fees that they charge. It makes using a crowdfunding platform more trustworthy as all transfers are traceable and transparent. The inherent security and decentralisation of the blockchain protects against malicious actors and single points of failure.

- It solves the problem of storage on the blockchain being expensive by storing info that is not crucial to fund transfers to IPFS which saves on gas without sacrificing security or decentralisation.

- Once deployed on a blockchain smart contracts can not be edited or changed so therefore must be free of bugs. This problem was avoided by extensive unit testing and end-to-end testing of all areas of the project.

- The cost of functions in smart contracts can increase with usage if not implemented efficiently. This can lead to the use of contracts being prohibitively expensive as they scale in use. This problem has been avoided by rigorous scalability testing to ensure that gas usage for all functions is linear.

## 5.2. Future work

The project could be developed in the future by adding some features and functionality. The user evaluation was helpful in pointing out desirable features that could be added in the future:

- Implement a rewards system for donators: This would involve having a system whereby donators can receive awards based on their donation amount. To do this would not be trivial but could potentially use Non-Fungible Tokens, (NFTs), to manage the transfer of ownership of digital assets that would be the rewards. You would need to implement this and the functionality without adding too much gas cost to the transactions. This could probably be a project in and of itself.

- Q&A / Comments section: Several users mentioned this as a possible feature. It would allow users to ask questions about specific projects and have them answered by the project’s creator. In a non-decentralised app this would be a trivial feature but for a fully decentralised app you would need to find the best way to store these comments without costing too much gas or implementing a traditional database and therefore sacrificing decentralisation.

- Project categories and a search function: Another feature that could be added would be to divide projects based on category i.e., tech, design etc. and have projects be searchable. This would involve finding a decentralised way of storing the projects separately and only calling the ones you need while also being able to only pull the info of the ones you need from the blockchain. You would have to do all this without adding to much gas cost.