CS 491: Charity Tracker

Chad Saltzman

21 November 2021

Contents

Abstract

Charity fraud is a common issue where organizations masquerade themselves as a valid charity, but instead take and use the charity donations for their self-interest. In these situations, it is very difficult for charities to prove their authenticity to new donators. Currently the only way to check the authenticity of a charity is by verifying if the charity is registered with the government and has no fraud claims with the Federal Trade Commission. Applying blockchain technology to charities would resolve this issue by allowing charities to publicly display the supply chain process of all donations proving their legitimateness. Potential Donators can observe past donations and even new donations identifying what the contribution was used for or if it is still pending an application. The introduction of blockchain would create more transparency and encourage new donators to trust and commit to contributing to the proposed cause.

Introduction

The Charity Tracker Application is designed to protect benefactors from contributing money to non-ethical charities and increase the reputation of charities. This is done through full transparency of the charity where most of the information related to the charity is publicly accessible allowing people to perform their own investigations on the validity of the charity. The blockchain technology allows for this transparency to occur when every donation is processed as a transaction and is verified and added to the blockchain. With the blockchain being publicly viewable, contributors will be allowed to track what their donation has been used for and/or verify the integrity of the charity. Smart Contracts will be used to integrate with the blockchain to update statuses and retrieve information about the charity and donations.

Description

The Charity Tracker application was designed using the programming languages python and solidity. Python was used for the core application and Solidity was implemented to incorporate smart contracts on the blockchain. To run this application, one must first deploy the smart contract. The parameters to deploy the smart contract is going to be an array containing the names of all the charities, and then another array holding the addresses of those charities. Once deployed, the contract address and the abi of the Ethereum Virtual Machine must be copied and inserted into python file to link the smart contract to the source code. Once these steps have been completed the application can be ran by executing the command python3 CharityTracker.py

Python Code Details

Libraries

As seen in Figure 1, the libraries used in this application are json, web3, sys, and getpass. The json library is used to parse the abi and convert it into a python dictionary. The web3 library is used to interact with both the Ganache network and the smart contracts. Sys is implemented to end the program when the user has finished accessing the application. Getpass is used as a secure way to retrieve the private key from the user for their Ethereum account.

```
import json
from web3 import Web3
import sys
from getpass import getpass
```

Figure 1: This figure displays the libraries that were imported.

Variables

Figure 2 shows the static variables that are used multiple times throughout the application. These variables are ganache_url, address, web3, contract, blockchainOptions and charityList. The ganache_url variable is used by the Web3 library to create a web3 object linked to the ganache Ethereum account which contains the many accounts used. The web3 object allows for interaction between the ganache network and the smart contracts deployed. The contract variable holds the address to the deployed smart contract. BlockchainOptions is a list containing the different choices that the user can select including “Complete a donation” and “View the status of a donation”. The charityList variable holds the details for each charity including the charity names and their respective addresses. In the future I would like to make a smart contract to retrieve the charity details so that it is not explicitly available in the source code.

```
ganache_url = "http://127.0.0.1:7545"
    web3 = Web3(Web3.HTTPProvider(ganache_url))
    abi =
json.loads('[{"inputs":[{"internalType":"string[]","name":"CharityName","type":"s
tring[]"},{"internalType":"address[]","name":"CharityAdd","type":"address[]"}],"s
tateMutability":"payable","type":"constructor"},{"inputs":[{"internalType":"uint2
56","name":"num","type":"uint256"},{"internalType":"string","name":"trx","type":"
string"},{"internalType":"string","name":"CharityName","type":"string"}],"name":"
contributeDonation","outputs":[],"stateMutability":"nonpayable","type":"function"
},{"inputs":[{"internalType":"address","name":"UserAddress","type":"address"}],"n
ame":"getBalance","outputs":[{"internalType":"uint256","name":"","type":"uint256"
```

```
}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"string
","name":"searchCharityName","type":"string"}],"name":"getCharityAddress","output
s":[{"internalType":"address","name":"tempAdd","type":"address"}],"stateMutabilit
y":"view","type":"function"},{"inputs":[{"internalType":"string","name":"trx","ty
pe":"string"}],"name":"getStatus","outputs":[{"internalType":"string","name":"sta
tus","type":"string"}],"stateMutability":"view","type":"function"},{"inputs":[{"i
nternalType":"string","name":"searchCharityName","type":"string"}],"name":"getTot
alDonated","outputs":[{"internalType":"uint256","name":"runningTotal","type":"uin
t256"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"s
tring","name":"newStatus","type":"string"},{"internalType":"string","name":"trx",
"type":"string"}],"name":"updateStatus","outputs":[],"stateMutability":"nonpayabl
e","type":"function"}]')

    address =
web3.toChecksumAddress("0xe0d7A1Ae367010F6Ea2512300f2196c70cB64A69")  # Deployed
Contract's address
    contract = web3.eth.contract(address=address, abi=abi)  # Creates an object
for the contract.

    charityList = {
        "Feeding America": "0x4c2b2972B316d77Ad6C2be9AAa6AbCd845FFB38a",
        "American Red Cross": "0x5D42f081Af27233Ab7ad8fE7039c6B145440d055",
        "St. Jude Children's Research Hospital":
"0xDB57c099b403C62388Dfd6a5E5e02fCE83a656E3"
    }
    blockchainOptions = ["Complete a donation", "View the status of a donation",
"View the total amount donated", "Update Status", "Verify Charity Address",
"Choose a different charity", "Exit"]
```

Figure 2: This figure displays the core variables used throughout the application.

Operations

The first choice the user is given is what charity they would like to select. Once selecting the charity, the user can always go back and switch between the provided charities, but after selecting one, all future operations will be directly correlated to the charity selected. An example of the charity selection screen is provided in Figure 3.

```
Choose a charity from the list below:
1. Feeding America
2. American Red Cross
3. St. Jude Children's Research Hospital
```

Figure 3: This figure displays the options of charity provided to the user.

After selecting a charity, the user will be prompted with a list of operations as detailed in Figure 4. For each operation the user will be providing additional information to fullfill the operation detailed such as providing donation amount, charity name, or the transaction address that they are looking for a status on. After the user has completed any operations that they would like, they are able to exit the application by inputting the value 7 into the terminal which will close the application.

```
      Welcome to the Feeding America Charity


choose from the following operations:
1. Complete a donation
2. View the status of a donation
3. View the total amount donated
4. Update Status
5. Verify Charity Address
6. Choose a different charity
7. Exit
```

Figure 4: This figure displays the different operations that the user can choose to interact with a charity

Complete a donation

The contribute a donation operation gathers the donation amount, user's Ethereum wallet address, and their private key to process the transaction. Once the information is gathered a transaction is executed from the user's account to the charity account for the amount specified. The "tx" variable specifies the transaction data and the "signed_tx" variable is the execution of the transaction on the blockchain with the transaction and user's signature. Once the transaction is completed, the user will be given a transaction hash which can be used to check the status of their donation. Finally, a call transaction to the smart contract must be executed to transfer the donation data to the contract to account for the statuses of the donation and the total donation of the charity selected as shown in Figure 5.

```
      donationValue = int(input("Enter in the value of your donation (Ether):"))

            account = input("Enter in your Ethereum Address: ")
            priv_key = getpass("enter in your private key: ")
            nonce = web3.eth.getTransactionCount(account)
            tx = {          #Need to verify the donater has enough ether to
donate.
                'nonce': nonce,
                'to': charityList[list(charityList)[charityValue - 1]],
                'value': web3.toWei(donationValue,'ether'),
                'gas': 2000000,
                'data': b"",
                'gasPrice': web3.toWei('50', 'gwei')
            }

            signed_tx = web3.eth.account.signTransaction(tx, priv_key)
```

```
            tx_hash =
web3.toHex(web3.eth.sendRawTransaction(signed_tx.rawTransaction))
            print(f"You can track the status of this donation with the following
hash code: {tx_hash}")
            print(web3.toWei(donationValue,'ether'))
            contract.functions.contributeDonation(web3.toWei(donationValue,'ether
'), tx_hash,list(charityList)[charityValue - 1]).transact()
```

Figure 5: This figure displays the source code that executes completing a transaction and logging the transaction data into the smart contract.

View the status

The view status operation allows the user to input the transaction hash returned from their donation and retrieve the status of that donation. The status right after the donation will be set to "Initial Donation" and from there can be updated by the charity administrators. A call to the smart contract is performed to retrieve this information as shown in figure 6.

```
tx = str(input("What is the transaction hash of your donation: "))
            status = contract.functions.getStatus(tx).call()
            print(f"the status of the transaction {tx} is: {status}")
```

Figure 6: this figure displays the source code that retrieves the status of a donation

View the total amount donated

The view total amount donated operation allows the user to see the total contributions to the charity selected. This is retrieved through a call to the smart contract which stores this information as shown in Figure 7.

```
balance = contract.functions.getTotalDonated(list(charityList)[charityValue -
1]).call()
            print("\nTotal donated = " + str(web3.fromWei(balance, "ether")) + "
ether")
```

Figure 7: this figure displays the source code that retrieves the total amount donated to the selected charity.

Update Status

The update status operation allows the status of the donation to be updated. Currently anyone can update the status of a donation which is a current flaw in the project. In the future I would like to implement an administrator role with executive privileges and only users with the administrator role can make this change. The change is completed through a transaction with the smart contract as shown in Figure 8.

```
tx = input("Enter in the hash associated with the donation you would like to
update: ")
            status = input("What is the updated status for this transaction: ")
            tx_hash = contract.functions.updateStatus(status,tx).transact()
            print(f"Status has been updated. The transaction hash for this is:
{tx_hash}")
```

Figure 8: This figure displays the source code that updates the status of a donation.

Verify Charity Address

The verify charity address operation returns to the terminal the Ethereum address associated with the charity so that the user can verify the validity of this address and look at the chain of transactions performed to and from this account. Therefore, if the status of their donation shows as delivered, then they should be able to see the transaction leaving the charity's account and it going to a new account. Figure 9 details the process of retrieving the address of the charity from the smart contract.

```
address = contract.functions.getCharityAddress(list(charityList)[charityValue -
1]).call()
            print(f"The ethereum address of {list(charityList)[charityValue - 1]}
is: {address}")
```

Figure 9: This figure displays the source code that retrieves the address of the selected charity.

Switch Charities

The switch charities operations return the user back to the main menu where they get to select from the list of predefined charities. Once the user selects one of the charities, they are returned to the operation menu where they can interact with that charity.

Exit

The exit operation terminates the application.

Solidity Code Details

Structures

Within the solidity smart contract, three structures are created to store the data for the Contributor, Charity, and the status of a donation. The Status structure will keep track of the status of each transaction so that the status can then be changed later. The Contributor structure is used to hold the address and the total amount donated by a single person. The Charity details structure holds the details differentiating each charity including the charity Ethereum address, charity name, and the total amount donated to the charity.

```
struct Status{
    string donationAddress;
    string status;
}

struct Contributer {
    address personalAddress;
    uint256 totalDonated;
}

struct CharityDetails {
    address charityAdd;
    string charityName;
    uint256 totalDonation;
}
```

Figure 10: This figure displays the structures created for the solidity smart contract.

Variables

Figure 11 shows the static variables used throughout the smart contract. The statuses array is a holds multiple instances of the status structure. The charities array holds multiple instances of the charity details structure. Contributors holds multiple instances of the contributor structure. The charityAddresses array holds the addresses associated to the charities inputted when deploying the smart contract.

```
Status[] statuses;
CharityDetails[] charities;
Contributer[] contributers;
address[] private charityAddresses;
```

Figure 11: This figure displays the static variables used throughout the solidity code.

Constructor

When the smart contract is deployed the constructor function is called. The constructor function, as seen in Figure 12, creates a new instance of CharityDetails structure and adds it to the array of all charitydetails. It initializes the total donation to zero and correctly adds the charity name and charity address. Along with this, the charity address is added to the list of all charity addresses.

```
constructor(string[] memory CharityName, address[] memory CharityAdd) payable {
    require(CharityName.length == CharityAdd.length, "Number of Charities is not equal to the number of addresses");

    for (uint256 i = 0; i < CharityAdd.length; i++){
        charityAddresses.push(CharityAdd[i]);
    }
    for (uint256 i = 0; i < CharityName.length; i++){
            charities.push(CharityDetails({
            charityAdd: CharityAdd[i],
            charityName: CharityName[i],
            totalDonation: 0
        }));
    }
}
```

Figure 12: This figure displays the constructor function which is ran when the smart contract is deployed.

Contribute Donation

The contributeDonation function, as shown in Figure 13 is used to update the smart contract after a donation to a charity is performed. When a donation is completed, the total donation amount for that charity is updated by the amount donated along with the individuals' personal details being added into a new instance of the contributors structure. Once the new instance is created, their personal total amount donated is equal to the donation amount contributed. To keep track of the status of the donation, a new instance of the status structure is created with the transaction address being set as well as the status being set to "Initial Donation" by default.

```
function contributeDonation(uint256 num, string memory trx, string memory CharityName) public
{
 /*  address contributeradd = msg.sender;
   bool found = false;
   require(contributeradd.balance >= num, "Unfortunately there are insufficient funds for a donation of that size");
   */
   for (uint256 i = 0; i < charities.length; i++){
        if (keccak256(bytes(charities[i].charityName)) == keccak256(bytes(CharityName))){
            charities[i].totalDonation += num;
        }
    }


   address contributeradd = msg.sender;
   bool found = false; // Resets found variable for next check
   for (uint256 i = 0; i < contributers.length; i++){
       if (contributers[i].personalAddress == contributeradd){
           found = true;
           break;
       }
   }

   // creates a new contributer if it does not exist already
   if (found == false){
       contributers.push(Contributer({
           personalAddress: contributeradd,
           totalDonated: 0
       }));
   }
   // Adds donation ammount to contributers total donated
    for (uint256 i = 0; i < contributers.length; i++){
       if (contributers[i].personalAddress == contributeradd){
           contributers[i].totalDonated += num;
           break;
       }
    }
    // creates inital status
    statuses.push(Status({
           donationAddress: trx,
           status: "Initial Donation"
       }));
```

Figure 13: This figure displays the contribute Donation function which adds all transaction data to the smart contract.

Update Status

The update status function, as seen in Figure 14, updates the status of a transaction. Once a transaction has been completed and the next stage of transporting the donated money to the designated cause is complete, the status will be updated by a charity administrator. The update status function works by finding the status instance with the transaction hash provided and changes the status variable to the new status string that was also passed in. If the transaction was not found in a instance of the status structure then a new instance is created for that transaction with the new status.

```
// Takes a transaction address and status value as input and updates the status of the transaction
 function updateStatus(string memory newStatus, string memory trx) public {
    bool found = false;
    for (uint i = 0; i < statuses.length; i++) {
        if(keccak256(bytes(statuses[i].donationAddress)) == keccak256(bytes(trx))){
            statuses[i].status = newStatus;
            found = true;
            break;
        }
    }
    if (found == false){
        // 'Status({...})' creates a temporary
        // Status object and 'statuses.push(...)'
        // appends it to the end of 'statuses'.
        statuses.push(Status({
            donationAddress: trx,
            status: newStatus
        }));
    }
}
```

Figure 14: This figure displays the update status function which changes the status of a donation

Get Functions

The getStatus, getCharityAddress, getBalance, and getTotalDonated functions, as seen in Figure 15, all return the value specified in the function name. For the get balance function it simply returns the balance of the address making the call to the function. For the other functions, they loop through all instances of the respective structure storing the required data and when the proper instance is found it will return the value specified.

```
function getStatus(string memory trx) public view
        returns (string memory status)
{
    bool found = false;
    for (uint i = 0; i < statuses.length; i++) {
        if(keccak256(bytes(statuses[i].donationAddress)) == keccak256(bytes(trx))){
            status = statuses[i].status;
            found = true;
        }
    }
    require(found, "the transaction hash was not found");
}

function getCharityAddress(string memory searchCharityName) public view
        returns (address tempAdd)
{
    for (uint i = 0; i < charities.length; i++){
        if (keccak256(bytes(charities[i].charityName)) == keccak256(bytes(searchCharityName))  ){
            tempAdd = charities[i].charityAdd;
        }
    }

}

function getBalance(address UserAddress) public view
        returns (uint256)
{
    return UserAddress.balance;

}

function getTotalDonated(string memory searchCharityName) public view
        returns (uint256 runningTotal)
    {
        runningTotal = 1;
        for (uint256 i = 0; i < charities.length; i++){
            if (keccak256(bytes(charities[i].charityName)) == keccak256(bytes(searchCharityName))  ) {
                runningTotal = charities[i].totalDonation;
            }
        }
    }
```

Figure 15: This figure shows the getter functions which return their respective value from the smart contract.

How to Run

To run this application first one must deploy the Charity.sol smart contract file in remix. To deploy the smart contract, one must first compile then pass in the <list of charities>, <list of charity addresses> to deploy. An example for this would be ["Feeding America","American Red Cross"], ["0x6Fc2B9B82538AB4388c469D012d979606f40220B","0xb85aa5e1e334597f12fF70419fA373BFe998a210"]. Once the smart contract is deployed you must copy the smart contract hash from remix and paste it into the code as the address as show in Figure 16. For my example I used ganache to create sample Ethereum wallets and to do this one must do so in remix by selecting the Web3 Provider Environment and inputting the appropriate address. Once the contract address is updated, the python file can be ran using the command python3 CharityTracker.py.

Figure 16: This figure displays updating the contract address in the python code.

References

1. https://web3py.readthedocs.io/en/stable/quickstart.html
2. https://docs.soliditylang.org/en/v0.8.10/
3. https://www.youtube.com/watch?v=p5W67zTQFRM&t=627s&ab_channel=DappUniversity