Title: Food banks Transparency and Traceability

Figure a strategy to assure the donors that the food banks have actually used the donations for the majority of decisions

Venkata Bala Vamsi Atukuri, vatukuri@buffalo.edu, 50469056
Sasidharreddy Lankireddy,slankire@buffalo.edu,50469217

Abstract:
To create a blockchain application that aggregates donors and food banks where donors, rather than traditional food banks, make the crucial decision of how to use the funds raised, thereby reducing the risk of fraud. Donors have access to the Foodbank's past effectiveness on the platform with complete transparency, fostering trust. Blockchain is used to verify and validate payments, collect donor votes, and verify food bank purchase bills. Ultimately we aim to standardize data collection and track data as it passes through an established system. With all of this information readily available, choices can be made to create a system that is more effective.

Issues addressed :
Every year, about 9 million people die from hunger. Food banks play a significant part in providing food to the poor, but it is still unclear how the money is used transparently. Although there are similar measures at the level of a single bank. We plan to address the transparency and traceability of all food banks by using blockchain to donate money and allowing donors to vote for specific food items based on the shelf life of the products, cost, etc. listed by the food bank. The platform's donors certify the entire process, including the use of donations for food purchases and expenses, which has an effect on how well-regarded food banks are as a result. We leverage smart contracts to store the payment transactions, donor votings and donor validations of foodbank expenses.

Steps 1&2: The donor selects a food bank in the first phase mostly based on its reputation and dependability.

Step 3: A smart contract is used to facilitate the payment transaction between the food bank and the donor.

Step 4: Verification and validation of payment transactions take place through blockchain.

Step 5: After the validations, the transactions are entered into a distributed and decentralised ledger.

Step 6: The selected foodbank proposes multiple plans to all the donors and lists all the costs and terms associated with each and every plan.

Step 7: The donors on the network vote for the plan. verification and validation of donor votes take place all the votes are stored in the decentralised ledger.

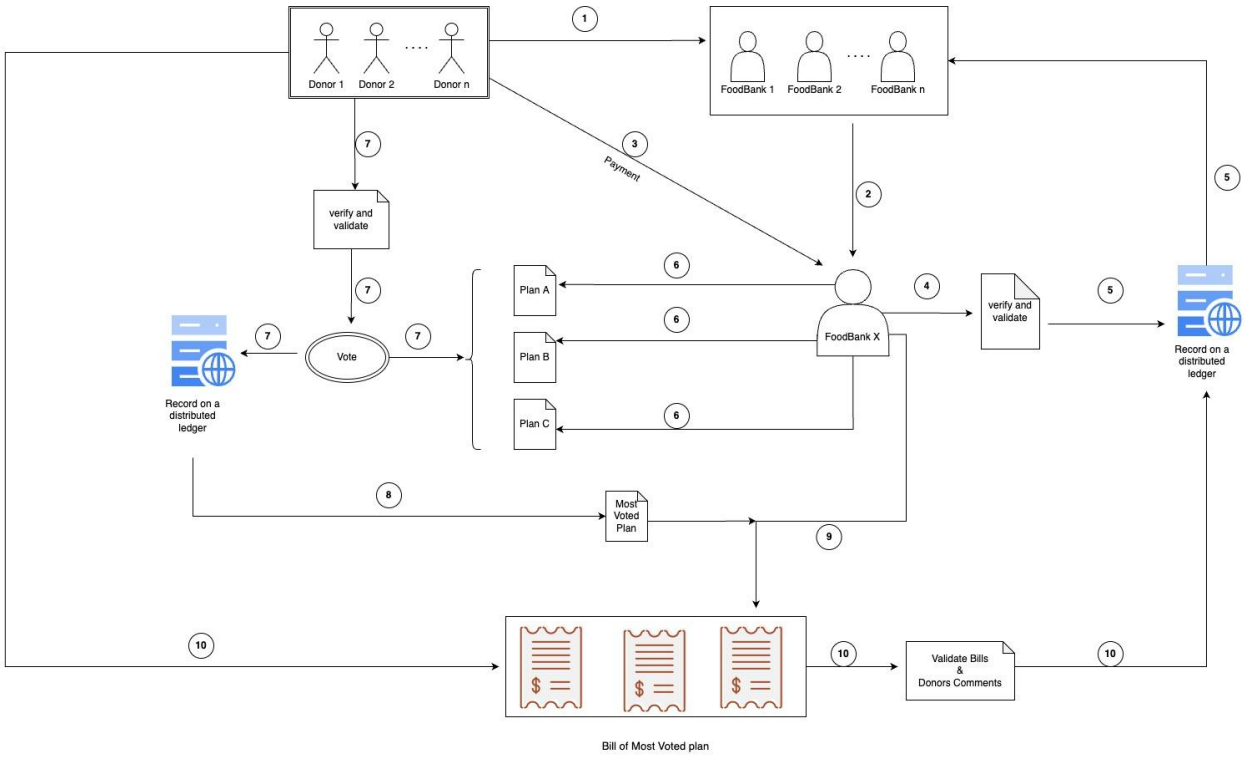

Fig: Flow chart indicating the entire process step by step.

Step 8: Foodbank selects the most voted plan from the network.

Step 9: The foodbank starts utilizing the funds and produces all the purchase orders according to the plan voted by the donors.

Step 10: The network's donors verify the purchase order and bills, as well as whether the foodbank adhered to the majority-approved plan. If not, the food bank gets a bad review from the donors, which lowers its rating.

Decentralised Application Overview:

The application will have mainly three users, the food bank, donors and the application deployer we call him the ChairPerson. The application is built using Next.js a React framework for the

front end and smart contracts for the backend.The entire data of the application is decentralised and stored in On chain Memory and can be accessible by every user which enables the trust and transparency.
We have deployed the entire project into four smart contracts to cater the data storage needs and Interfaces in smart contract to interact with the smaller contracts that supplement the project data needs.The agile development mechanism has been adopted while creating the application and divided the development into four parallel phases.

However, before building the application we had put in a thorough effort to distinguish how to define constraints on how our project distinguishes from the systems that exists currently.For this we have constructed a quad chart diagram illustrating the advantages that overcome the problems faced in the current system.

Usecase: Foodbanks Transparency and Traceability (FTT) **Problem Statement:** Figure a strategy to assure the donors that the food banks have actually used the donations for the majority of decisions.	**Issues with existing centralized model:** 1. Inefficient and non-transparent utilisation of funds. 2.Each and every food bank acts as a centralised system and barely had the visibility on how funds are spent. 3. Less visibility on the funds usage and donations and improper evaluation metrics for a food bank. 4. Donor doesn't have the option to choose on what to donate. 5. No model for automatic payment settlement a lot of interventions are in place for the existing model.
Proposed blockchain-based solution: 1. Foodbanks make their data transaparent by enabling block chain which is visibe to every end user. 2. The donors tend to vote on the plan that needs to implemented by the food bank. 3.Improves the transparency and enables healthy competetion among food banks, 4. Seamless payment settlement is enabled. 5. All transactions recorded on blockchain on a distributed ledger the food banks cannot manipulate these transactions.	**Benefits:** 1. Improves the visibility and make the system efficient. 2. Better customer experience 3. Helps in efficient utilisation of donor funds. 4.Oppurtunity to follow majority donation on the donation process. 5. Seamless payment settlement system 6. Better management of resources. 7. Potential for creating a open community with high trustworthiness.

Fig 1: FTT quad chart: use case, issues, blockchain solution, and benefits

Stage 1:Building the Entities(Users):

The main entities of this project are donors and foodbanks. Donor responsibilities include donation to the food banks and voting for the best plan that suites their interest.The first step in building these entities is their creation. Once all the smart contracts are deployed we need the deployer of the smart contract i.e application developer to enter the address of smart contracts on the admin page and then we have a open ecosystem of foodbanks and donors who can be listed on the homepage.A user can register as a foodbank as well as donor we have two separate methods for the creation.For registering a donor we use a modifier to check whether the donor is valid or not and then the donor is created. The same method follows for the food bank and also some global variables are created to handle the data of the smart contracts.

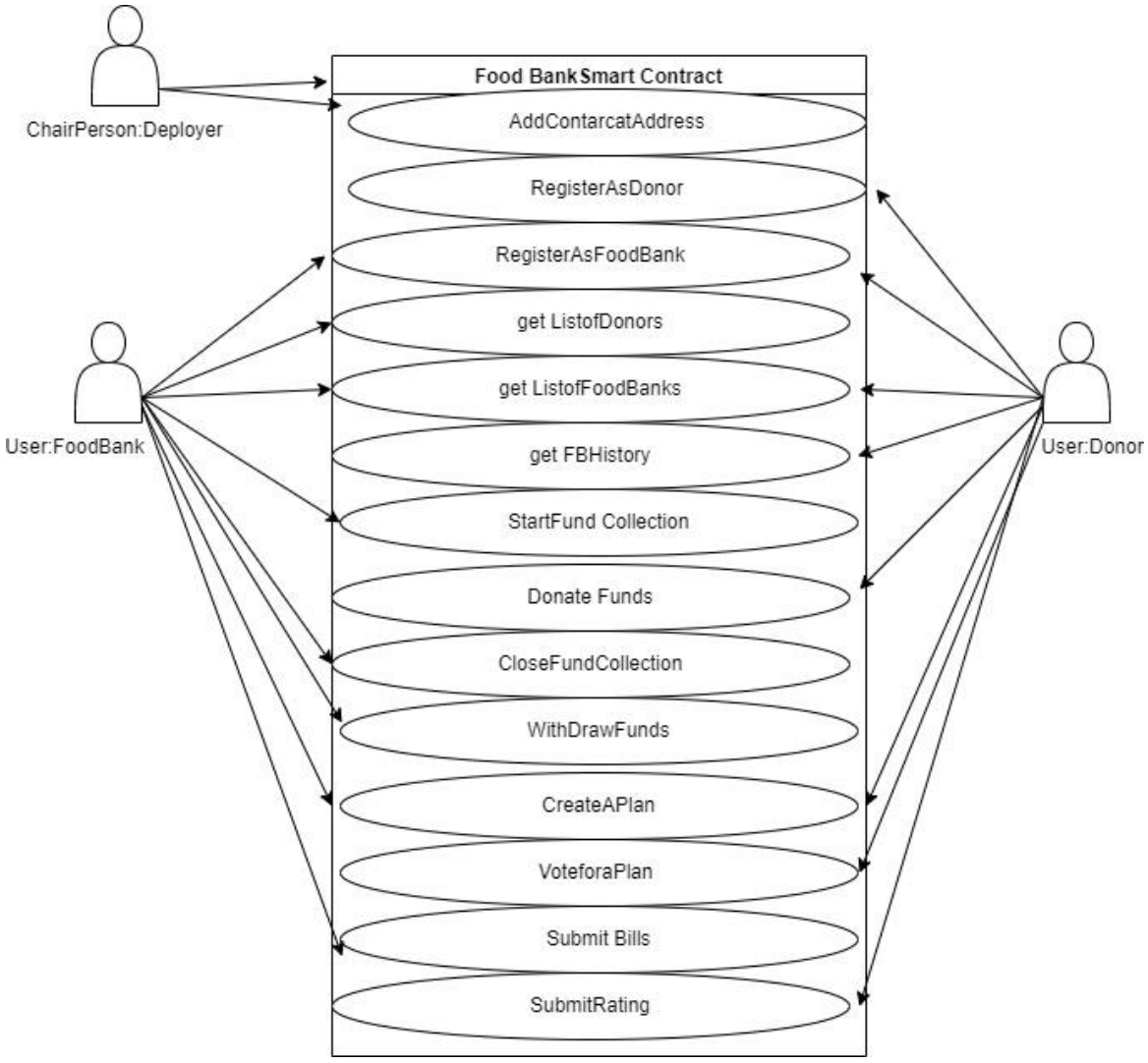

Fig 2: Use Case Diagram Illustrating the main users and methods of Smart Contract

We mainly have 4 smart contracts the first contract defines the food bank which is pillar of this project and inherits minor methods from plan,rating and voting smart contracts. The plan smart contract have a plan and mapping of plan id to its plan and struct of plan which have multiple items and amount allocated corresponding to each of the item.we have three major methods that get and create and update the plan created by food banks.It also has the method for getting the most voted plan.And in the rating contract we have rating structure and methods to calculate the ratings. And in the voting contract we have the structure of the vote and a create vote method that sets the interface for user to vote.In the foodbank smart contract we have major methods that define the entire project which are mentioned below in the use case diagram.

The major methods are only mentioned in the Use case and the sequence diagram inorder to preserve the clear understanding to the instructor and all these methods are part of the Foodbank SmartContract.

Stage 2:Creating the Variables and Modifiers:

We have created the structural variables that each inherits certain variables. We created struct implementations for FoodBank, FoodBank History and a map with foodbank_Id to foodbank Struct and Struct of DonorActicity and DonorDetails and Plans and other variables. All the structural variables inherit the entire variables mentioned in each and every structure. And we have the address of the smart contract that needs to be added by the deployer of the smart contract. To restrict the user privileges we have used Functional Modifiers.

Function modifiers are used to change or restrict the behavior of a function in a smart contract. We can use a modifier to automatically check a condition prior to executing the function. For example before executing a function to create a donor account we need to verify the donor already exists in the system or not. Modifiers are reusable blocks of code that can be attached to one or many function in your contract.

We have established modifiers at each and every step inorder to check the privileges of the users(A donor cannot perform the tasks that neeed to be done by foodbank and viceversa).Modifiers play a vital role in maintaining the stability of a smart contract by checking the modifiers before executing the functions. And methods play a crucial role in implementing the functional capabilities in the smart contract. In order to get a clear picture of the smart contract we have drawn a contract diagram that illsutsrates the variables, structure variables and access modifiers and methods. The key modifiers and methods were explained in the implementation section.

Foodbank.sol

```
struct FBank { string name; uint rating; address payable acc_address; uint total_money_raised;
               bool active; uint[] history }
mapping(address => FBank) foodbanks;
address[] foodbank_addresses;

struct FBankHistory { uint id; address payable fb_addr; string name; bool active; uint curr_step;
                      uint total_steps; uint money_raised; uint rating address[] donors; uint[] donor_acts;
                      uint[] plans; uint[] ratings; uint most_voted_plan_id; uint[] votes}
uint latest_fb_hist_id;
mapping(uint => FBankHistory) fbankshistories;

struct DonorActivity { uint id; address donor_acc_addr; address food_bank_addr; uint fb_hist_id;
                       bool funded; uint funded_amount; bool voted; uint voted_for_plan_id}
mapping(uint => DonorActivity) donor_activities
uint latest_donor_activity_id;

struct Donor { acc_address; string name; uint total_amount_funded; uint[] activity}
mapping(address => Donor) donors;
address[] donor_addresses;

address VotingContract;
address RatingContract;
address PlanContract;
address chairperson;
```

```
modifier onlyCDeployer();
modifier validForFbCreation(address acc_addr) ;
modifier onlyChair(address payable acc_addr, address payable fb_acc_addr){};
modifier validFBHistStep(uint hist_id, uint req_step){};
modifier onlyChairFBHist(uint hist_id, address user_addr){};
modifier validDonor(address donor_address){};
modifier vaidForDonorCreation(address acc_addr){};
modifier onlyValidDonor(uint hist_id, address user_address){};
modifier validPlanId(uint plan_id, uint hist_id){};
```

```
function _onlyCDeployer() private view{};
function setVotingContractAddr(address voting_addr) onlyCDeployer() external{};
function setRatingContractAddr(address rating_addr) onlyCDeployer() external{};
function setPlanContractAddr(address plan_addr) onlyCDeployer() external{};
function getListOfFBS() public view returns(address[] memory, string[] memory, bool[] memory, uint[] memory, uint[] memory){};
function createFB(address payable acc_addr, string memory name) public validForFbCreation(acc_addr){};
function createFBHistory(address payable acc_addr, string memory name, address payable fb_acc_addr) onlyChair(acc_addr, fb_acc_addr) public{};
function getFBHistory(address acc_addr) public view returns(string[] memory, uint[] memory, bool[] memory, uint[] memory, uint[] memory){};
function viewFBHistory(uint fb_hist_id) public view returns(address, string memory, bool, uint[] memory, uint[] memory, uint[] memory){};
function _validFBHistStep(uint hist_id, uint req_step) private view {}
function _onlyChairFBHist(uint hist_id, address user_addr) private view {};
function _validDonor(address donor_address) private view{};
function start_fund_collection(address user_addr, uint hist_id) public validFBHistStep(hist_id, 0) onlyChairFBHist(hist_id, user_addr){};
function donate_funds(uint hist_id) public validFBHistStep(hist_id, 1) validDonor(msg.sender) payable{};
function withdraw_funds(address user_addr, uint hist_id) public onlyChairFBHist(hist_id, user_addr) validFBHistStep(hist_id, 3){};
function createDonor(address acc_addr, string memory name) public vaidForDonorCreation(acc_addr){};
function getListOfDonors() public view returns(address[] memory, string[] memory, uint[] memory){};
function viewDonorDetails(address donor_addr) public view returns(string memory, uint, uint[] memory){};
function getDonorActivityDetails(uint donor_activity_id) public view returns(address, uint[] memory, bool, address, string[] memory){};
function updateFbHistStateToStartFundCollection(address user_addr, uint hist_id) public onlyChairFBHist(hist_id, user_addr) validFBHistStep(hist_id, 0){};
function createFbHistPlan(address user_addr, uint hist_id, string[] memory item_names, uint[] memory item_amounts) public onlyChairFBHist(hist_id, user_addr)
validFBHistStep(hist_id, 1){};
function startVoting(address user_addr, uint hist_id) public onlyChairFBHist(hist_id, user_addr) validFBHistStep(hist_id, 1){};
function registerVote(uint plan_id, uint hist_id, address user_addr) public validPlanId(plan_id, hist_id){
function endVotingAndFetchMostVotedPlan(uint hist_id, address user_addr) public validFBHistStep(hist_id, 2) onlyChairFBHist(hist_id, user_addr){};
function submitBill(uint hist_id, address user_addr) public validFBHistStep(hist_id, 4) onlyChairFBHist(hist_id, user_addr){};
function submitRatingUtil(uint hist_id, address user_addr, uint rating) private{};
function submitRating(uint hist_id, address user_addr, uint rating) public validFBHistStep(hist_id, 5) onlyValidDonor(hist_id, user_addr){};
```

Fig 3: Contract diagram illustrating the variables,modifiers,methods.

Stage 3:Creating the methods:

For creating the methods we have followed a step by step method the first step of the smart contract is to update the address of a interface contract's that are used in the code.For this we created three different methods for updating the addresses of these three smart contracts.And once the above method is done, We have the ecosystem set.We then created the methods to create the donor and foodbank. We have used access modifiers infront of these methods to check that the donor and foodbank haven't existed before.

Once the Food banks and donors created then a foodbank needs to update the state inorder to collect the funds, for this method the foodbank needs to update the state and should start the fund collection.Once the foodbank start the fund collection the donors can donate the funds. For this we have donate funds methods that donates certain amount of funds to the smart contract.Once the donations are done the foodbank can stop accepting the funds.Once food bank stops accepting the funds it creates the plans in which it wants to spend the donated funds.The foodbank lists multiple plans and requests the donors to vote.We have register Vote method for performing this in the smart contract.

Now that the donors have requested to vote the donor needs to chose the plan for choosing the suitable plan. Once all the user votes the foodbank will close the voting and show the most voted plan.Once the most voted plan is selected the Food bank can withdraw the funds and submit the bills corresponding to the fund utilisation.Once the user sees the bills the user can rate the foodbank on the basis of their service and trustworthiness.All the methods in the system helps in preforming the steps in harmony.

Stage 4:Integrating the Smart Contract with the front end:

We have simultaneously integrated each and every method of the smart contract to the font-end using next.js during the development process.All the react pages are integrated using the smart contract address and the ABI that we got from the Remix IDE. We updated the ABI after each and every step of development of the smart contract.Once the methods are implemented the return types are taken and are displayed on the UI.

We have sequence diagram to illustrate the interactions,since we havent used any external Database the entire data is stored in Blockchain. Each and every method of the smartcontract either intercats with the smart contracts or obtain the data from the smart contract.The methods that returns the data from the smart contract returns only data from the smart contracts these methods donot perform any of the operations.However the methods interacting with the users perform the useractions based on the method that is called.

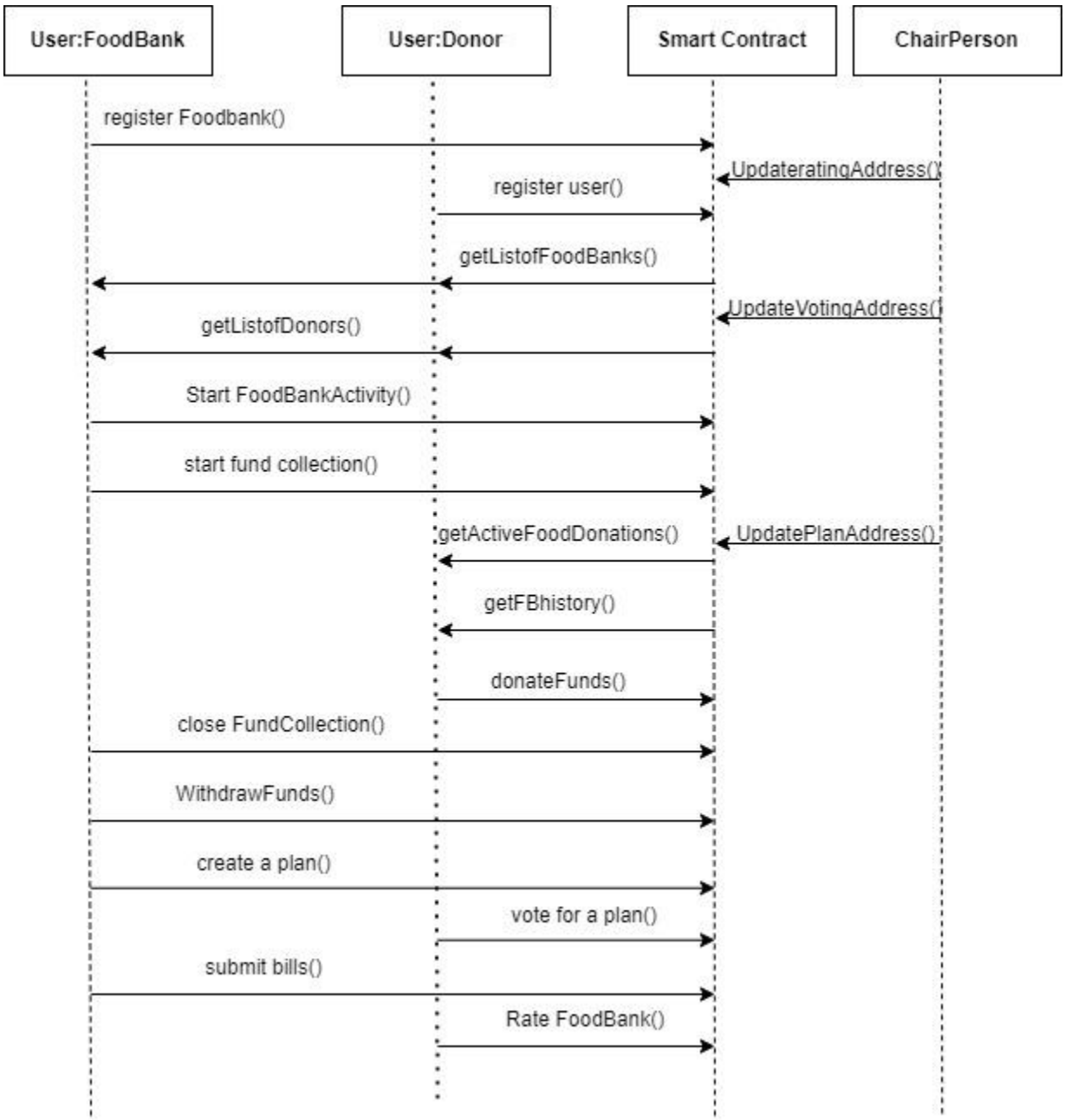

Fig 4: Sequence Diagram Illustrating the main users and interactions with the systems.

Smart Contract Functionality:

Structural Implementations:

The foodbank struct has name variable and uint of rating that is given by user and and address corresponding to the foodbank address and total_money_raised as money raised by the foodbank and bool active gives the current fundraising status of the foodbank and uint of histories that represent the ids of foodbank history.

```
struct FBank {
        string name; // Foodbank name
        uint rating; // overall rating of the fb
        address payable acc_address;  // account address of the foodbank
        uint total_money_raised; // sum of all funds raised thus far
        bool active; // raising any funds currently or not.
        uint[] history;//Food Bank's history including active dispensement
    }
```

The foodbank history struct has id variable and address corresponding to the foodbank address and name on which the fundraising is done and bool active gives the fundraising status of the foodbank and uint of total_steps involved in the process and money_raised for the particular event and rating on that event and address of all the donors donated for the event and actions of donors and plans they voted for and ratings for the user and most_voted_plan_id for that particular plan.

```
struct FBankHistory{
        uint id;
        address payable fb_addr;
        string name;   // eg: Coronovirus fund
        bool active;
        uint curr_step;
   // { 0: 'nothing', 1: 'fb has started fund collection'}
        uint total_steps;
        uint money_raised;
        uint rating;
        address[] donors;
   // who are all the donors who donated to this particular dispensement.
        uint[] donor_acts;
        uint[] plans;
   // list of all plans given by foodbank
        uint[] ratings;
```

```
        uint most_voted_plan_id;
        uint[] votes;
    }
```

We have donor with address as acc_address and name of the string and total_amount_funded and array of the user activity corresponding to the donor.

```
 struct Donor{
        address acc_address;
        string name;
        uint total_amount_funded;
        uint[] activity;
    }
```

We have donor activity with id on the basis of the registered number and address as don_acc_address of doonor and food_bank_addr the address to which the foodbank is donated fb_hist_id that returns the data of the transaction and a boolean variable of funded and total_funded_amount in that activity and check whether the user had voted or not and id of the plan for which the donor has voted for.

```
struct DonorActivity{
      uint id;
      address donor_acc_addr;
      address food_bank_addr;
      uint fb_hist_id;
      bool funded;
      uint funded_amount;
      bool voted;
      uint voted_for_plan_id;
    }
```

We have a struct plan which has plan_item_ids array containing all the plans and amount collected on the each plan.

```
struct Plan{
        uint id;
        uint[] plan_item_ids;
        uint collected_amount;
   // summation of donor's donated amount for this particular plan
    }
```

We have planitem which has id and item_name and amount_allocated for each of the item.

```
struct PlanItem{
        uint id;
        string item_name;
        uint amount_allocated;
    }
```

We have rating struct which has id of the rating and the hist_id and value which the donor has given and address and donor_acc_addr and donor_donation_amount that the donor has donated.

```
struct Rating{
        uint id;
        uint hist_id;
        uint value; // rating value
        address donor_acc_addr;
        uint donor_donation_amount;
    }
```

For every Vote struct we have id and donor_address and voted_for_plan_id and amount_donated for the corresponding vote.

```
struct Vote{
        uint id;
        address donor_address;
        uint voted_for_plan_id;
        uint amount_donated;
    }
```

Modifiers:

The onlyCDeployer() validates whether the message sender is the chairperson or not and then executes the method of updation of addresses.

```
function _onlyCDeployer() private view{
      require(msg.sender == chairperson);
    }


     modifier onlyCDeployer()
     {
       _onlyCDeployer();
       _;
     }
```

The ValidFBHiststep checks whether the foodbanks current step is equal to the steps that is required to proceed.

```
 function _validFBHistStep(uint hist_id, uint req_step) private view {
       FBankHistory memory fb_hist = fbankshistories[hist_id];
       require(fb_hist.curr_step == req_step);
    }

    modifier validFBHistStep(uint hist_id, uint req_step){
        _validFBHistStep(hist_id, req_step);
       _;
    }
```

The modifier ValidForFbcreation checks whether the acc_addr is present, if at all the account address is present that means the foodbank already exists and the method isn't implemented.

```
modifier validForFbCreation(address acc_addr)
     {
        bool not_present = true;
        for (uint i = 0; i < foodbank_addresses.length; i++) {
            if (foodbank_addresses[i] == acc_addr) {
                not_present = false;
                break;
            }
        }
        require(not_present == true);
       _;
     }

    function createFB(string memory name) public
validForFbCreation(msg.sender){
       address payable acc_addr = payable(msg.sender);

       foodbanks[acc_addr].name = name;0o
       foodbanks[acc_addr].acc_address = acc_addr;
       foodbanks[acc_addr].active = false;
       foodbanks[acc_addr].total_money_raised = 0;
       foodbanks[acc_addr].history = new uint[](0);
       foodbank_addresses.push(acc_addr);
```

Key Methods:

Donate_funds: called when a donor is donating funds

- The function donate funds takes hist id as an argument and utilizes written modifiers to validate the sender account address and foodbank history's current step.
- We update the donation amount in both foodbank history and foodbank.
- We also create a record in donor activities indicating that the donor made a donation to this specific food bank history.
- We also append the donor address and donor activity id to the foodbankhistory donors address and foodbankhistory donor acts arrays.

```
function donate_funds(uint hist_id) public validFBHistStep(hist_id, 1)
validDonor(msg.sender) payable{
        address donor_addr = msg.sender;
        uint donation_amount = msg.value;

        // updating donation amount in fundbank and fundbank hist
        fbankshistories[hist_id].money_raised =
fbankshistories[hist_id].money_raised + donation_amount;
        address foodbank_addr = fbankshistories[hist_id].fb_addr;
        foodbanks[foodbank_addr].total_money_raised =
foodbanks[foodbank_addr].total_money_raised + donation_amount;

        // updating in donor and donor activity
        donors[donor_addr].total_amount_funded =
donors[donor_addr].total_amount_funded + donation_amount;
        DonorActivity memory don_act = DonorActivity({ id:
latest_donor_activity_id + 1, food_bank_addr: foodbank_addr, fb_hist_id:
hist_id, funded: true,
                                                        funded_amount:
donation_amount, voted: false,  voted_for_plan_id: 0, donor_acc_addr:
donor_addr});
        latest_donor_activity_id = latest_donor_activity_id + 1;
        donors[donor_addr].activity.push(latest_donor_activity_id);
        donor_activities[latest_donor_activity_id] = don_act;

        // updating foodbank_hist

fbankshistories[hist_id].donor_acts.push(latest_donor_activity_id);
        fbankshistories[hist_id].donors.push(donor_addr);
    }
```

Withdraw_funds: called when foodbank wants to withdraw funds from the smartcontract.

- Using modifiers, we determine whether the message sender is the one who created the foodbank and whether the current step of the foodbank history is valid to withdraw funds or not.
- We obtain total amount raised by the foodbank for that history and transfer it to the message sender account address.
- We also increase the foodbank_history current_step, preventing money from being withdrawn again.

```
 function withdraw_funds(uint hist_id) public onlyChairFBHist(hist_id,
msg.sender) validFBHistStep(hist_id, 3){
        address payable fb_addr = fbankshistories[hist_id].fb_addr;
        uint amount_collected = fbankshistories[hist_id].money_raised;
        fb_addr.transfer(amount_collected);
        fbankshistories[hist_id].curr_step = 4;
    }
```

Register Vote: called when the donor tries to cast the vote

- Using modifiers, we first determine whether the plan id to which the donor is attempting to vote is valid and belongs to that specific foodbank history.
- We also verify whether or not the donor has funded.
- We also look to see if the donor has previously voted for any of the foodbank's plans.
- In the corresponding donor activity, we also capture the plan id and increment the collected amount for that plan.
- We also create a record in the votes map indicating that this specific donor voted for this specific plan.

```
    function registerVote(uint plan_id, uint hist_id) public
validPlanId(plan_id, hist_id){
       address user_addr = msg.sender;

       uint donor_activity_id = 0;
       for(uint i = 0; i < fbankshistories[hist_id].donor_acts.length;
i++){

if(donor_activities[fbankshistories[hist_id].donor_acts[i]].donor_acc_addr
== user_addr){
                donor_activity_id = fbankshistories[hist_id].donor_acts[i];
                break;
```

```
            }
        }

        require(fbankshistories[hist_id].curr_step == 2);
        require(donor_activity_id != 0, "Donor hasn't funded");
        require(donor_activities[donor_activity_id].voted == false,
"Already voted");

        // updating donor_activity
        donor_activities[donor_activity_id].voted = true;
        donor_activities[donor_activity_id].voted_for_plan_id = plan_id;
        // uint donated_amount =
donor_activities[donor_activity_id].funded_amount;

        // updating plan collected amount
        InterfacePlan plan_interface = InterfacePlan(PlanContract);
        plan_interface.updatePlanCollectedAmount(plan_id,
donor_activities[donor_activity_id].funded_amount);

        // creating a new vote and updating in fb_hist
        InterfaceVoting voting_interface = InterfaceVoting(VotingContract);
        uint latest_vote_id = voting_interface.createVote(user_addr,
plan_id, donor_activities[donor_activity_id].funded_amount);
        fbankshistories[hist_id].votes.push(latest_vote_id);
    }

    function endVotingAndFetchMostVotedPlan(uint hist_id) public
validFBHistStep(hist_id, 2) onlyChairFBHist(hist_id, msg.sender){
        uint[] memory mem_plans =  fbankshistories[hist_id].plans;

        InterfacePlan plan_interface = InterfacePlan(PlanContract);
        uint max_voted_plan_id =
plan_interface.fetchMostVotedPlan(mem_plans);

        assert(max_voted_plan_id != 0); // Max Voted Plan doesn't exists

        fbankshistories[hist_id].curr_step = 3;
        fbankshistories[hist_id].most_voted_plan_id = max_voted_plan_id;
    }
```

Storage Overview:

The data is stored in the blockchain in the mentioned data structure as follows:

FoodBank.Sol:

address[] foodbank_addresses: contains the addresses of the registered foodbanks on the platform.

mapping(address => FBank) foodbanks: A map that maps food_bank_address to FBank structure.

mapping(uint => FBankHistory) fbankshistories: A map that maps food_bank_history_id to FBankHistory structure.

uint latest_fb_hist_id: Latest food bank history id registered on our platform.

address[] donor_addresses: contains the addresses of the registered donors on the platform.

mapping(address => Donor) donors: A map that maps donor address to Donor structure.

uint latest_donor_activity_id: Latest Donor Activity registered on out platform

mapping(uint => DonorActivity) donor_activities: A map that maps donor_activity_id to DonorActivity.

address VotingContract: Voting smart contract address

address RatingContract: Rating smart contract address.

address PlanContract: plan contract address.

address chairperson: Address of the person who has deployed the smart contract.

PlanC.sol:

uint latest_plan_id: latest plan_id registered on our platform

mapping(uint => Plan) plans: A map that map plan_id to Plan structure.

RatingC.sol:

uint latest_rating_id: latest rating_id registered on our platform.

mapping(uint => Rating) ratings: A map that maps rating_id to Rating Structure.

Voting.sol:

uint latest_vote_id: latest vote_id registered on our platform.

mapping(uint => Vote) votes: A map that maps vote_id to Vote Structure.

How to start Dapp:

Once you are in foodbank_dapp_site directory in the terminal:

- **npm install**: to install necessary packages
- **npm run dev:** to start the project

Smart Contract Implementation:

Deployment:

We used the truffle to deploy all of the smart contracts in the ganache local test network. The addresses of the other smart contracts we are using as interfaces were then copied and pasted into the chairperson methods from the Frontend admin screen. Before adding the smart contracts to the front end code, we compiled them and created ABI for them in remix.

- ***truffle compile***: to compile all the smart contracts
- ***sudo truffle deploy*****:** to deploy the contracts onto local ganache network.

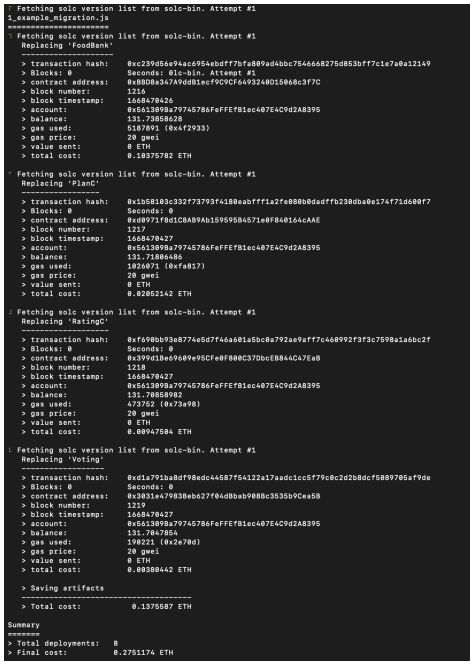

Fig5: Figure illustrating the implementation of smart contract.

Working:

We have attached the working screenshots attached below.

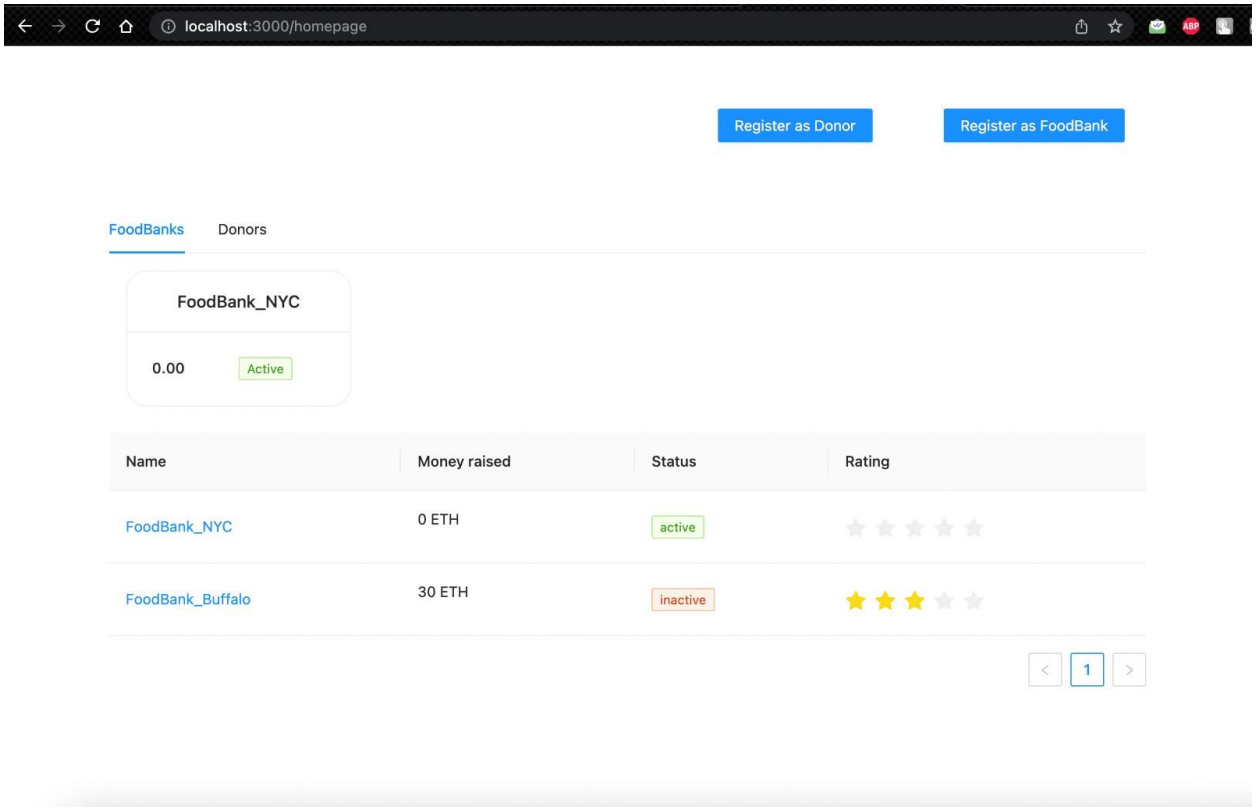

Fig6: UI of the foodbank page with active fund collection.

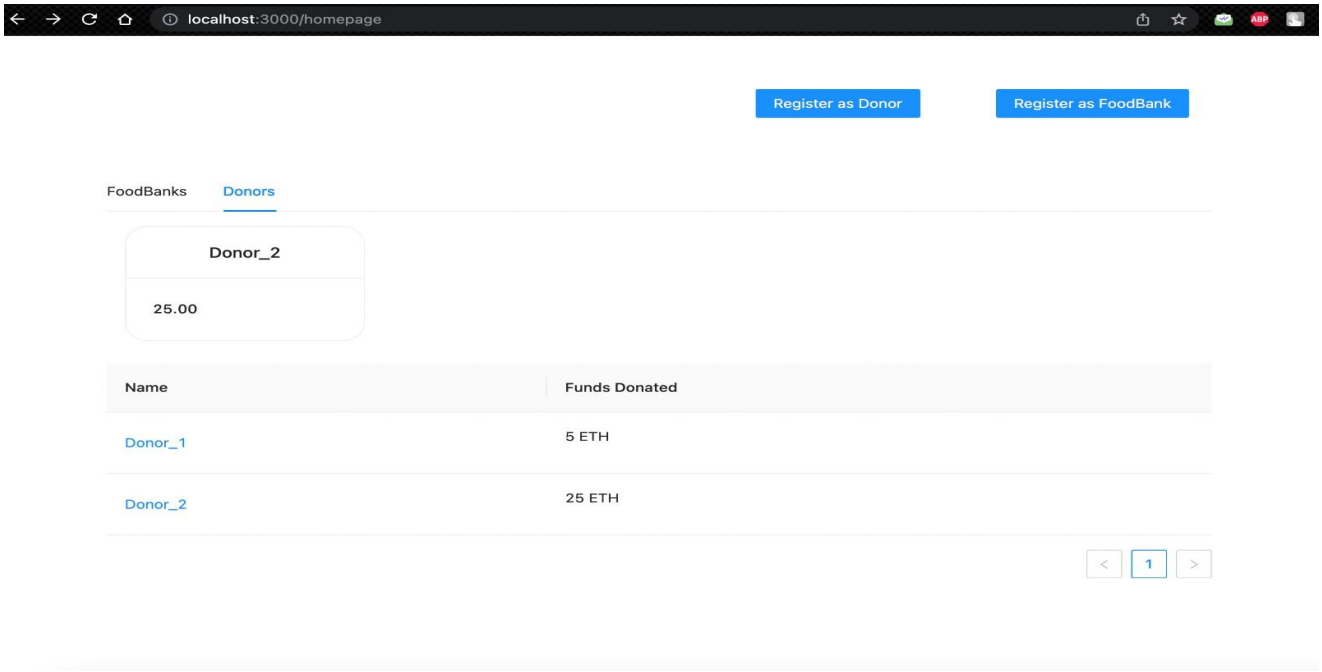

Fig7: UI of the donor page after donating the funds.

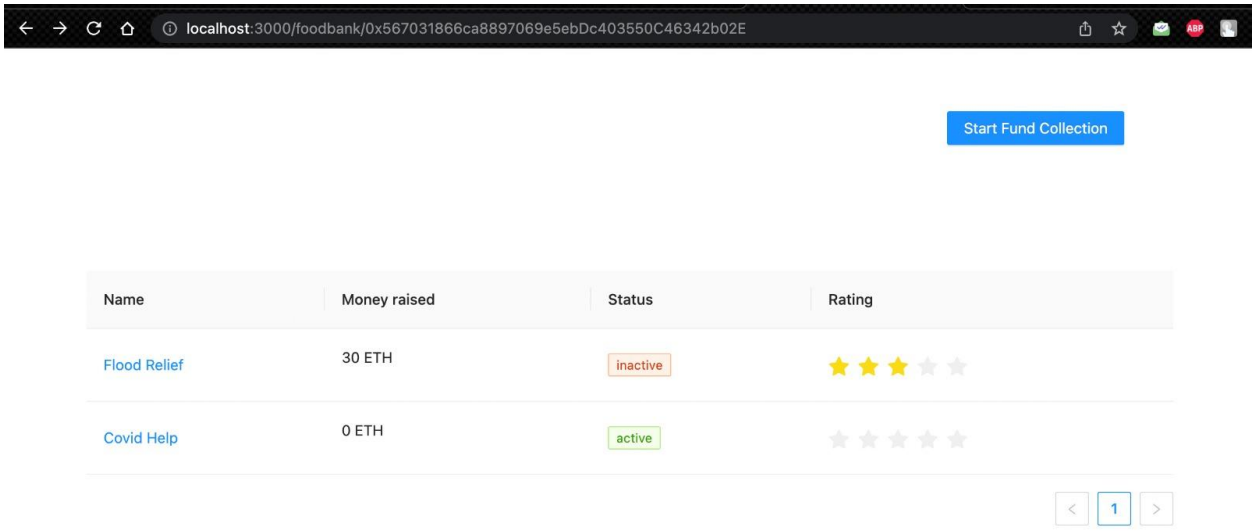

Fig 8: UI of the foodbank page after fundraising and rating process.

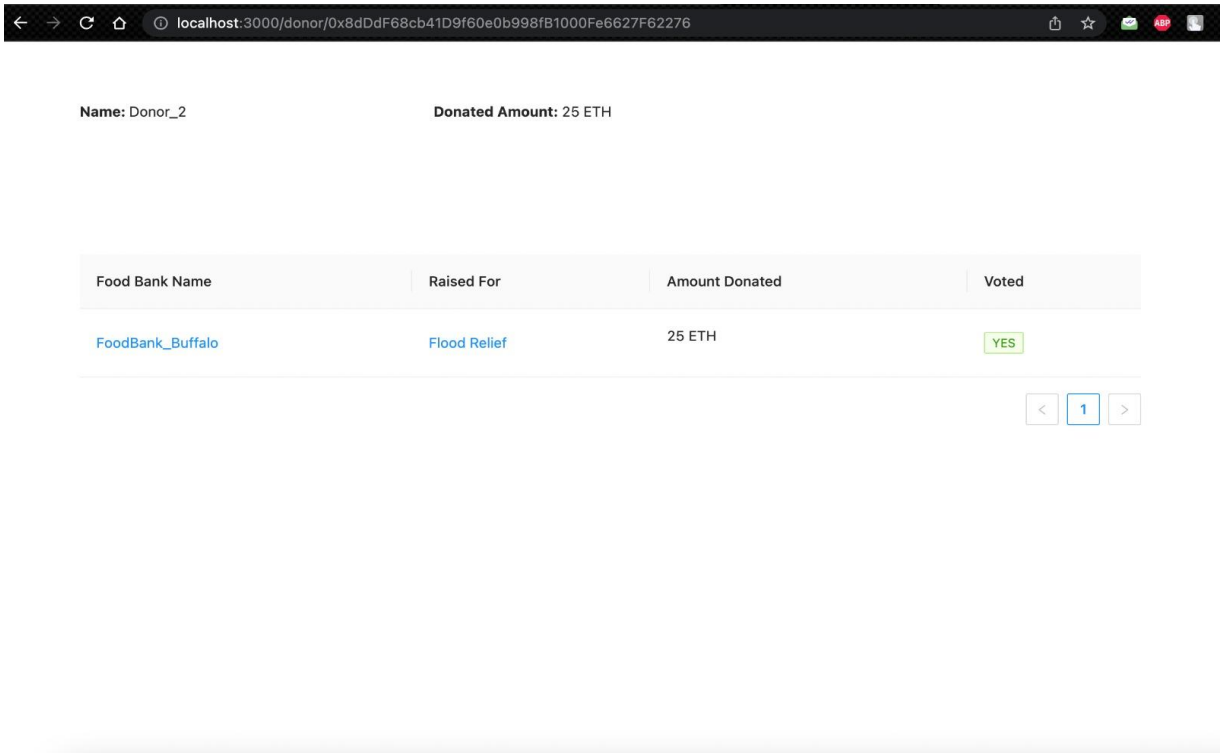

Fig 9: UI of the dono rpage after voting process.

Testing the Dapp with test cases:

We then tested the dapp for the test cases using the truffle test command we had tested for the following methods

1.Creating a Foodbank.
2.Creating a foodBank History.
3.Creating a Donor
4.Updating the FoodHistory to start Fund collection.
5.Creating a new Plan.
6.Checking the voting system.

```
vamsi@Balas-MacBook-Air Project % truffle test
Using network 'development'.


Compiling your contracts...
===========================
> Compiling ./contracts/foodBank.sollc-bin. Attempt #1
> Artifacts written to /var/folders/gs/ds_4fdkj6ks66pz4flr91yx00000gn/T/test--1590-QVmVK37ed1wc
> Compiled successfully using:
   - solc: 0.8.17+commit.8df45f5f.Emscripten.clang
" Fetching solc version list from solc-bin. Attempt #1

  Contract: FoodBankion list from solc-bin. Attempt #1
    ✔ should create FoodBank correctly (293ms)tempt #1
    ✔ should create FoodBank History correctly (260ms)
    ✔ should create donor correctly (138ms) Attempt #1
    ✔ update FbHistory to start fund collection (234ms)
    ✔ Should create a new plan correctly (668ms)mpt #1
    ✔ Start voting should work correctly (187ms)mpt #1


  6 passing (2s)
```

Fig 10: Running test cases in terminal.

Phase-III

Token Creation(ERC/721):

We are using ERC-721 token for this project and we have used openzepplin framework for writing FbToken smart contract.Once the token is minted we store a mapping named **userOwnedTokens** in the smart contract that has user_address to token_ids, and this map is updated every time a token is issued to or utilised by a user.

```
function safeMint(address to, string memory uri) internal{
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();
    _safeMint(to, tokenId);
```

```
        _setTokenURI(tokenId, uri);
        userOwnedTokens[to].push(tokenId);
}
```

Token Usage:

We are rewarding the ERC-721 tokens named BART to the user who has rated the foodbank after the donation,once these are rewarded the donor can use the following tokens in the subsequent donations,each token has a weight of 2 Voting ETH for the next voting.The token doesn't have a fiat conversion value but however the user can make a case in the follow-through donations for choosing the most voted plan. For example, if there exists two persons on the platform A and B. Let's say A is more frequent donor and B is new to the platform. If A has donated on the platform he has tokens that can help him make a case while choosing the most selected plan.However, A cannot control the system with altogether because token's are transferred to the chairperson of the contract once they are used.However, user can hold these tokens as symbol of appreciation.

Note: In the case of free market scenarios,the token value is decided by the monetary value,demand and supply and various factors effecting its purpose.However for this project we have considered the value to 2 Voting ETH (BART) and is minted after a user rating the foodbank.The chairperson of the contract will receive all the tokens once these are utilised and these tokens will stay under the smart contract.

```
  function deduct_tokens(uint no_of_tokens) external{
    address user_addr = msg.sender;
    uint256[] memory original_token_ids = userOwnedTokens[user_addr];
    uint original_tokens_arr_size = original_token_ids.length;

    require(original_tokens_arr_size >= no_of_tokens);

    for(uint i = 0; i < no_of_tokens; i++){
      uint token_id = original_token_ids[i];
      safeTransferFrom(user_addr, chairperson, token_id);
      userOwnedTokens[chairperson].push(token_id);
    }
```

```
    uint256[] memory new_token_ids = new
uint256[](original_tokens_arr_size - no_of_tokens);
    for(uint i = no_of_tokens; i < original_tokens_arr_size; i++){
      new_token_ids[i - no_of_tokens] = original_token_ids[i];
    }
    userOwnedTokens[user_addr] = new_token_ids;
  }
```

Code explanation: Front end calls the function deduct tokens, passing the number of tokens(i.e n) to be deducted as an argument. The smart contract initially checks if the user has as least the number of tokens requested to be deducted. After that, n tokens would be transferred from the user to the chairperson. We also update the mapping of userOwnedTokens.

Token Smart Contract Implementation:

This contract FBToken, which implements the ERC721 contract and the ERC721URIStorage contract,uses the Ownable contract from OpenZeppelin module of solidity and the Counters contract from OpenZeppelin. It has an interface called InterfaceFBToken, with mainly two functions: rewardToken and fetch_balance. It also has a mapping called userOwnedTokens, which maps addresses to an array of token IDs.

The constructor sets up the token name and symbol and increments the token ID counter. The rewardToken function mints a token to a given address. The fetch_balance function returns the balance of tokens for a given address. The deduct_tokens function deducts tokens from a given address and transfers them to the chairperson. .

Steps to deploy the smart Contract on Goerli Test Network:

*sudo truffle deploy --network goerl*i

UI for Token Implementation:

Below are the snapshots for the implementation of tokens.The first snapshot illustrates the number of tokens after rating and and second image states the tokens during the time of donation.

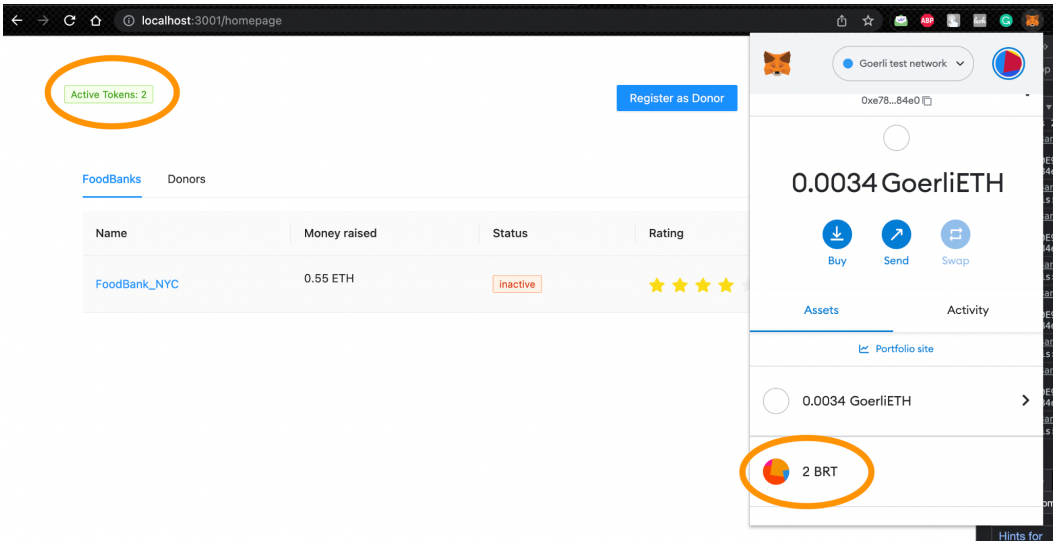

Fig 10: Illustrating the number of tokens present in donors account in Dapp.

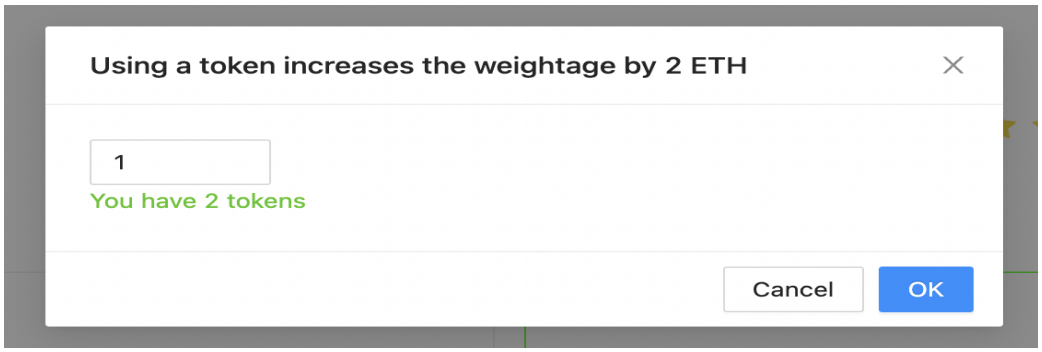

Fig 11: Illustrating the number of tokens present in donors while donating funds.

Tracking tokens on Test Network:
Inorder to track the token transactions over the Goerli Test Network we have attached the link below: [Link for Token Interactions on Goerli.]

Instructions to deploy the token smart contract on Remix:
Step1: Remove the node modules directory present in the import statement of FB_token.sol file

I.e replace the following code block

```
import "../node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol";
import
"../node_modules/@openzeppelin/contracts/token/ERC721/extensions/ERC721URI
Storage.sol";
```

```
import "../node_modules/@openzeppelin/contracts/access/Ownable.sol";
import "../node_modules/@openzeppelin/contracts/utils/Counters.sol";
```

To

```
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
```

Step2: Deploy the Fb_token smart contract on remix using a VM.

Step3: Call rewardToken for rewarding the token to a user by changing the account address.

Step4: Call fetch_balance by passing address for validating whether the token is rewarded or not.

Step5: Call deduct_tokens with the address of user from whom account the tokens need to be deducted by passing the number of tokens.

Step6: Call fetch_balance by passing address for validating whether the token is rewarded or not.

Step7: Check the fetch_balance of chairperson of the smart contract to check whether the deduct_tokens are reflected to chairperson account or not.

Phase-IV:

Testing the SRapp on Goerli test network:

Step1: Go to the folder foodbank and run

npm install

truffle compile

***sudo truffle deploy --network goerl*i**

```
vamsi@Balas-Air foodbank % sudo truffle deploy --network goerli
Password:

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.
⠸ Fetching solc version list from solc-bin. Attempt #1
⠴ Fetching solc version list from solc-bin. Attempt #1
Migrations dry-run (simulation)
===============================
> Network name:    'goerli-fork'
> Network id:      5
> Block gas limit: 30000000 (0x1c9c380)


1_deploy_contracts.js
=====================
⠸ Fetching solc version list from solc-bin. Attempt #1
   Replacing 'FoodBank'
   --------------------
   > block number:        8099848 solc-bin. Attempt #1
   > block timestamp:     1670529949
   > account:             0x7a62350709Cd106038B9746F04AfB9d40813Dd4C
   > balance:             0.500049953718912318
   > gas used:            5327830 (0x514bd6)
   > gas price:           2.500089717 gwei
   > value sent:          0 ETH
   > total cost:          0.01332005299692411 ETH

⠴ Fetching solc version list from solc-bin. Attempt #1
   Replacing 'PlanC'
   -----------------
   > block number:        8099849 solc-bin. Attempt #1
   > block timestamp:     1670529949
   > account:             0x7a62350709Cd106038B9746F04AfB9d40813Dd4C
```

Step 2: Go to the folder foodbank_dapp_site and run

npm install

npm run dev

```
vamsi@Balas-Air foodbank_dapp_site % npm run dev

> foodbank_dapp_site@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in 387 ms (159 modules)
```

Tracking the interactions on Blockchain:

Below are the links for the implementation of all the smart contracts.Please find the tracking of the each of the link mentioned below.

FoodBank and Donor Interactions: etherscan link

Plan details Interactions: etherscan link

Rating Interactions: etherscan link

Voting Interactions: etherscan link

Token Interactions: etherscan link

References:

1. Ramamurthy B. *Blockchain in Action / Bina Ramamurthy.* 1st edition. Manning Publications; 2020.

2. *In world of wealth, 9 million people die every year from hunger, WFP Chief tells Food System Summit | World Food Programme,* https://www.wfp.org/news/world-wealth-9-million-people-die-every-year-hunger-wfp-chief-tells-food-system-summit,2022

3. *What Does Blockchain Have to Do With Food Banking? - Food Bank News*, Chris Constazo, https://foodbanknews.org/what-does-blockchain-have-to-do-with-food-banking.