Group C: Mini Project Report

Title:

Develop a Blockchain based application for transparent and genuine charity

Objective:

- **Improve Transparency:** Record all donations on the blockchain, enabling anyone to validate contributions and monitor total donations in real-time.

- **Boost Confidence: Facilitate** direct communication between donors and the charity, eliminating intermediaries and minimizing risks of fraud or mismanagement.

- **Guarantee Data Security:** Use blockchain's permanent and tamper-resistant nature to protect donation data from unauthorized changes.

- **Enhance Donation Experience:** Offer a simple and user-friendly platform for donors to easily create accounts, make donations, and follow their contributions.

Introduction:

The Blockchain-Based Charity Application is designed to enhance transparency, security, and trust in charitable donations. Traditional charity systems often suffer from issues like mismanagement, lack of transparency, and even fraud, making donors hesitant. By using blockchain technology, this project ensures that all donations are recorded immutably on the Ethereum blockchain, making them tamper-proof and fully traceable, which helps address these common challenges.

The application is built on a smart contract developed in Solidity, allowing donors to contribute directly without intermediaries. This eliminates the need for third parties, reducing the risk of funds being mismanaged. Additionally, both the total donations and individual donor details are accessible to everyone, promoting openness and accountability in the donation process.

By integrating Web3, the platform enables seamless communication between the frontend and blockchain, providing users with an easy-to-use interface for creating accounts, making donations, and tracking their contributions in real-time. This decentralized approach ensures a secure, transparent, and trustworthy environment for donations, fostering confidence among donors and stakeholders alike.

Requirements:

1. Software Requirements:

- Programming Languages: Solidity (for smart contracts), JavaScript (for frontend interaction).
- Blockchain Platform: Ethereum (using Ganache for local blockchain development).
- Development Environment: Remix IDE (for writing and deploying smart contracts), Visual Studio Code (for frontend development).
- Frameworks/Libraries: Web3.js (for connecting frontend with blockchain), React.js (for creating the frontend).

- MetaMask: Browser extension for enabling Ethereum transactions.
- Ganache: For running a local blockchain instance for testing.

2. Hardware Requirements:

- Processor: Intel i3 or above.
- RAM: 4GB minimum (8GB recommended for smooth development).

3. Blockchain Requirements:

- Ether (ETH): Required for gas fees during smart contract deployment and transactions on the Ethereum network.
- Test Network: A local Ethereum blockchain instance (Ganache) or test networks (Rinkeby, Ropsten) for testing purposes.

Implementation:

To install Node.js, follow these step-by-step instructions:

Step 1: Download Node.js Installer

1. Open your browser and go to the official **Node.js** website: https://nodejs.org/.
2. On the homepage, you will see two download options:
   - **LTS (Long-Term Support)** version: Recommended for most users and provides stability.
   - **Current** version: Includes the latest features but might not be as stable.

Choose the **LTS** version for a more stable experience.

Step 2: Download the Installer for Your Operating System

- Based on your operating system, the correct download will be suggested automatically (Windows, macOS, or Linux).
- Click on the download button to get the installer for your OS:
  - For **Windows**, the installer is an .msi file.
  - For **macOS**, it's a .pkg file.
  - For **Linux**, you might need to use the terminal for installation (more on that later).

Step 3: Run the Installer

For Windows:

1. Navigate to the folder where the .msi file was downloaded (usually the **Downloads** folder).
2. Double-click on the installer to run it.

3. Follow the installation wizard:
    - Accept the **license agreement**.
    - Choose the **installation directory** (or leave the default one).
    - Ensure that the option for **Automatically install the necessary tools** is selected.
4. Click **Next** and **Install**.
5. Once the installation is complete, click **Finish**.

For macOS:

1. Open the downloaded .pkg file.
2. Follow the instructions in the installer:
    - Accept the **license agreement**.
    - Choose the **installation path** (default path is fine).
3. Click **Install** and provide your macOS password when prompted.
4. Once the installation completes, close the installer.

For Linux:

1. Open a terminal window.
2. Install Node.js via your package manager based on your distribution.

Step 4: Verify Installation

After installation, verify that **Node.js** and **npm** (Node package manager) have been successfully installed by running the following commands in the terminal or command prompt.

For Windows:

- Open the **Command Prompt** by pressing Win + R, typing cmd, and hitting Enter.

For macOS/Linux:

- Open the **Terminal**.

Now, check the installed versions of Node.js and npm:

```
node -v
```

This command checks the Node.js version

```
npm -v
```

This command checks the npm version.

To install Ganache from SourceForge, follow these step-by-step instructions:

Step 1: Visit SourceForge.net

- Open your browser and go to SourceForge.net.
- In the search bar, type **Ganache** and press Enter.

Step 2: Find the Ganache Project

- In the search results, look for **Ganache** (by TruffleSuite).
- Select the appropriate result to visit the Ganache download page.

Step 3: Choose the Platform

- Once on the Ganache page, you'll see different download options based on your platform (Windows, macOS, or Linux).
- Click on the **Download** button for your operating system.

Step 4: Download the Installer

- The download will start automatically after clicking the download button.
- Wait for the download to complete. It will be an executable file (e.g., .exe for Windows, .dmg for macOS, or .AppImage for Linux).

Step 5: Install Ganache

- After the download is complete, navigate to the downloaded file in your system (usually in the "Downloads" folder).

For Windows:

- Double-click the .exe file to launch the installer.
- Follow the installation wizard steps (click "Next," accept the license agreement, and choose the installation path).
- Once the installation is complete, click **Finish**.

For macOS:

- Double-click the .dmg file to open it.
- Drag the Ganache application to the **Applications** folder.
- You may be prompted to allow the app to be installed from an "unverified developer." Approve it in **System Preferences > Security & Privacy**.

Step 6: Launch Ganache

- Once installed, open Ganache from the Start Menu (Windows), the Applications folder (macOS), or by double-clicking the .AppImage file on Linux.

To connect Ganache to MetaMask, follow these step-by-step instructions:

Step 1: Install MetaMask Extension

- First, if you don't already have **MetaMask**, install it from the browser extension store:
    - For **Chrome** or **Brave**, go to the MetaMask extension page.
    - For **Firefox**, go to the Firefox Add-ons page.
- Click **Add to Browser** and complete the installation.

Step 2: Open Ganache

- Launch **Ganache** from your desktop or application folder.
- When the Ganache workspace opens, it will automatically create a new Ethereum blockchain with several test accounts and 100 ETH in each account.
- Make sure Ganache is running.

Step 3: Retrieve Network Details from Ganache

- In Ganache, go to the **"Quickstart"** Ethereum workspace or create a new workspace.
- You'll see a screen displaying the **RPC Server** URL (usually something like http://127.0.0.1:7545) and the network ID (typically 5777).

Step 4: Open MetaMask and Add a Custom Network

- Open **MetaMask** in your browser.
- Click the MetaMask **icon** (top-right corner of the browser) to open the extension.
- If you are using MetaMask for the first time:
    - Create a new wallet or import an existing one using your secret recovery phrase.
    - Complete the setup.
- Once your wallet is set up, click on the **network dropdown** at the top, where it says "Ethereum Mainnet" or any other network you might be on.

Step 5: Add a New Custom Network in MetaMask

- From the network dropdown, click on **"Add network"** or **"Custom RPC"** (depending on your MetaMask version).
- You'll be taken to a page where you can add custom network details. Here's what you need to enter based on the Ganache info:
    1. **Network Name**: You can name it anything, such as **Ganache Localhost**.
    2. **New RPC URL**: Copy the **RPC Server** URL from Ganache (e.g., http://127.0.0.1:7545) and paste it here.
    3. **Chain ID**: Enter 5777 (or the chain ID shown in Ganache).

4. **Currency Symbol** (optional): You can put **ETH**.
5. **Block Explorer URL** (optional): Leave it blank.

- Once all the details are entered, click **Save** or **Add Network**.

Step 6: Switch to the Ganache Network in MetaMask

- After saving, MetaMask will automatically switch to your new **Ganache Localhost** network.

Step 7: Import Accounts from Ganache into MetaMask

- In the Ganache UI, you'll see multiple test accounts listed with their public addresses and private keys.
- In MetaMask, click the **Account Icon** (top-right corner) and select **Import Account**.
- In the **Private Key** field, paste the private key of any test account from Ganache (you can copy the private key by clicking the key icon next to the account).
- Click **Import**. Now, your MetaMask wallet will be connected to that specific Ganache account.

Step 8: Verify the Connection

- After importing the account, you should see the same **balance** of ETH (usually 100 ETH) in MetaMask as shown in the Ganache account.
- MetaMask is now connected to Ganache's local Ethereum blockchain!

Step 9: Start Using MetaMask with Ganache

- You can now use MetaMask to interact with the Ganache blockchain. For example, you can deploy smart contracts or test DApps connected to Ganache using MetaMask.

Develop a Blockchain based application for transparent and genuine charity, follow these step-by-step instructions:

1. Backend (Smart Contract) Setup

We'll start by configuring the smart contract with Truffle and Ganache.

Step 1: Install Required Tools

- Ensure you have **Node.js** and **Ganache** installed.
- Install **Truffle** globally:

```
npm install -g truffle
```

- Install **MetaMask** for interacting with the blockchain.

Step 2: Initialize Truffle Project

- Create a new folder for your project and initialize Truffle:

```
mkdir CharityBankApp
cd CharityBankApp
truffle init
```

step 3: Smart Contract Code

- Inside the contracts folder, create a new file CharityBank.sol and add the smart contract code you provided:

```
// SPDX-License-Identifier: MIT
pragma solidity >= 0.7.0;


contract CharityBank {


    // Structure to store donor's details
    struct Donor {
        string bankAccountNumber;
        string bankName;
        string donorName;
        uint donationAmount;
    }


    // Mapping to store the balance of each donor by address
    mapping(address => uint) private userAccount;


    // Mapping to check if an account already exists
    mapping(address => bool) private userExist;


    // Array to store all donors
    Donor[] private donors;


    // Total donations received
    uint public totalDonations;
    // Event to log donations
    event DonationReceived(string donorName, string bankName, string bankAccountNumber, uint amount);

    // Function to create an account for the donor
    function createAccount() public payable returns (string memory) {
        require(!userExist[msg.sender], "Account already exists!");
        userAccount[msg.sender] = msg.value; // Initialize user's account with initial funds
        userExist[msg.sender] = true; // Mark the user as having an account
        return "Account created successfully";
    }

    // Function to donate money
    function donate(string memory _bankAccountNumber, string memory _bankName, string memory _donorName, uint _amount) public payable returns (string memory) {
        require(userExist[msg.sender], "Account not created!");
        require(_amount > 0, "Amount must be greater than zero");
        require(msg.value == _amount, "Ether sent must match the donation amount"); // Ensure the correct amount of Ether is sent

        // Update the donor's account balance
        userAccount[msg.sender] += _amount;

        // Add to total donations
        totalDonations += _amount;

        // Log the donor's details
        Donor memory newDonor = Donor(_bankAccountNumber, _bankName, _donorName, _amount);
        donors.push(newDonor);
        // Emit an event for the donation
        emit DonationReceived(_donorName, _bankName, _bankAccountNumber,
_amount);

        return "Donation successful";
    }

    // Function to show the balance of the caller's account
    function accountBalance() public view returns (uint) {
        return userAccount[msg.sender];
    }

    // Function to view all the donations made so far
    function viewAllDonations() public view returns (uint) {
        return totalDonations;
    }

    // Function to view all donors and their details
    function viewAllDonors() public view returns (Donor[] memory) {
        return donors;
    }
}
```

Step 4: Configure Truffle for Ganache

- In truffle-config.js, Paste the following code

```
/**
 * Use this file to configure your truffle project.
 */

module.exports = {
```

```
/**
 * Networks define how you connect to your ethereum client and let you set the
 * defaults web3 uses to send transactions. If you don't specify one truffle
 * will spin up a managed Ganache instance for you on port 9545 when you
 * run `develop` or `test`. You can ask a truffle command to use a specific
 * network from the command line, e.g:
 *
 * $ truffle test --network <network-name>
 */

networks: {
  // Local Ganache development network
  development: {
    host: "127.0.0.1",     // Localhost for Ganache
    port: 7545,            // Ganache standard port
    network_id: "*",       // Match any network id
  },
},

// Set default mocha options here, use special reporters, etc.
mocha: {
  timeout: 100000  // Sets a reasonable timeout for tests
},

// Configure your Solidity compilers
compilers: {
  solc: {
    version: "0.8.21",  // Solidity version to compile your contracts
    settings: {
      optimizer: {
        enabled: true, // Enable optimizer to reduce gas costs
        runs: 200
      },
      evmVersion: "istanbul" // You can specify EVM version
    }
  }
},

// Truffle DB is disabled by default, can be enabled if needed
db: {
  enabled: false
}
};
```

Step 5: Compile and Deploy the Contract

1. **Compile** the contract:

```
truffle compile
```

2. Create a migration file under migrations called 2_deploy_contracts.js:

```
const CharityBank = artifacts.require("CharityBank");
module.exports = function (deployer) {
    deployer.deploy(CharityBank);
};
```

3. **Deploy** the contract:

```
truffle migrate --network development
```

2. Frontend Setup

Now let's create the frontend using React and Web3.js to interact with the deployed contract.

Step 1: Initialize React Application

Navigate to the root directory and create a frontend:

```
npx create-react-app client
```

```
cd client
npm install web3
```

Step 2: Import ABI

After the contract is deployed, the ABI (Application Binary Interface) will be available in the build/contracts folder.

- Copy CharityBank.json from build/contracts to client/src/contracts

Step 3: Frontend Code (React & Web3)

Now, inside the client/src folder, modify App.js to interact with the contract:

```
import React, { useState, useEffect } from 'react';

import Web3 from 'web3';

import CharityBank from './contracts/CharityBank.json';

import './App.css';



const App = () => {

  const [account, setAccount] = useState('');

  const [contract, setContract] = useState(null);

  const [donors, setDonors] = useState([]);

  const [totalDonations, setTotalDonations] = useState(0);

  const [bankAccountNumber, setBankAccountNumber] = useState('');

  const [bankName, setBankName] = useState('');

  const [donorName, setDonorName] = useState('');

  const [donationAmount, setDonationAmount] = useState('');



  useEffect(() => {
    const init = async () => {
        await loadWeb3();
        await loadBlockchainData();
    };
    init();
}, []);

const loadWeb3 = async () => {
    if (window.ethereum) {
        window.web3 = new Web3(window.ethereum);
        try {
            // Request account access if needed
            await window.ethereum.request({ method: 'eth_requestAccounts' });
            console.log('MetaMask detected and connected.');
        } catch (error) {
            alert('User denied account access.');
            console.error('User denied account access', error);
        }
    } else if (window.web3) {
        window.web3 = new Web3(window.web3.currentProvider);
        console.log('Legacy dapp browser detected. Connected.');
    } else {
        alert('Non-Ethereum browser detected. You should consider trying
        MetaMask!');
        console.error('No Ethereum provider detected');
    }
};


const loadBlockchainData = async () => {
    try {
        const web3 = window.web3;
        const accounts = await web3.eth.getAccounts();
        if (accounts.length === 0) {
            alert('No accounts found. Please connect MetaMask.');
            return;
        }
        setAccount(accounts[0]);
        console.log('Connected account:', accounts[0]);


        const networkId = await web3.eth.net.getId();
        console.log('Network ID:', networkId);
        const deployedNetwork = CharityBank.networks[networkId];
        if (!deployedNetwork) {
            alert('Smart contract not deployed to detected network.');
            console.error('Smart contract not found on network:', networkId);
            return;
        }
        const instance = new web3.eth.Contract(
          CharityBank.abi,
          deployedNetwork.address
        );
        setContract(instance);
        console.log('Contract instance set:', instance.options.address);

        // Load total donations
        const total = await instance.methods.viewAllDonations().call();
        setTotalDonations(total);
        console.log('Total Donations:', total);

        // Load donors
        const allDonors = await instance.methods.viewAllDonors().call();
        setDonors(allDonors);
        console.log('All Donors:', allDonors);
      } catch (error) {
        console.error('Error loading blockchain data:', error);
        alert('Error loading blockchain data. Check console for details.');
      }
    };

    const createAccount = async () => {
      if (contract) {
        try {
          if (!account) {
              alert("Account not found. Please connect MetaMask.");
              return;
          }
          const response = await contract.methods.createAccount().send({
              from: account,
              value: Web3.utils.toWei('0.01', 'ether') // Example value
          });
          console.log('Account creation response:', response);
          alert('Account created successfully!');
        } catch (error) {
          console.error('Error creating account:', error);
          alert('Error creating account: ' + error.message);
        }
      } else {
        alert("Contract is not loaded. Please try again.");
        console.error('Contract not loaded.');
      }
    };


    const donate = async (e) => {
      e.preventDefault();
      if (contract) {
```

```
try {

    // Validation checks

    if (!bankAccountNumber || !bankName || !donorName ||
donationAmount <= 0) {

        alert('Please fill in all fields with valid values.');

        return;

    }



    const donationAmountWei =
Web3.utils.toWei(donationAmount.toString(), 'ether');



    // Estimate Gas

    const gas = await contract.methods.donate(bankAccountNumber,
bankName, donorName, donationAmountWei).estimateGas({

        from: account,

        value: donationAmountWei

    });

    console.log('Estimated Gas:', gas);



    // Send transaction with estimated gas

    const response = await contract.methods.donate(bankAccountNumber,
bankName, donorName, donationAmountWei).send({

        from: account,

        value: donationAmountWei,

        gas
```

```
            });
            console.log('Donation response:', response);
            alert('Donation successful!');

            // Update the total donations and donors list
            const total = await contract.methods.viewAllDonations().call();
            setTotalDonations(total);
            const allDonors = await contract.methods.viewAllDonors().call();
            setDonors(allDonors);
        } catch (error) {
            console.error('Donation error:', error);
            alert('Donation failed! ' + error.message);
        }
    } else {
        alert("Contract is not loaded. Please try again.");
        console.error('Contract not loaded.');
    }
};

return (
    <div className="container">
        <h1>Charity Bank</h1>
        <p>Connected Account: {account}</p>
        <button onClick={createAccount}>Create Account</button>
```

```
<form onSubmit={donate}>
  <input
    type="text"
    placeholder="Bank Account Number"
    value={bankAccountNumber}
    onChange={e => setBankAccountNumber(e.target.value)}
    required
  />
  <input
    type="text"
    placeholder="Bank Name"
    value={bankName}
    onChange={e => setBankName(e.target.value)}
    required
  />
  <input
    type="text"
    placeholder="Donor Name"
    value={donorName}
    onChange={e => setDonorName(e.target.value)}
    required
  />
  <input
    type="number"
```

```
                placeholder="Donation Amount (in ETH)"
                value={donationAmount}
                onChange={e => setDonationAmount(e.target.value)}
                min="0.01"
                step="0.01"
                required
            />
            <button type="submit">Donate</button>
        </form>
        <h2>Total Donations: {Web3.utils.fromWei(totalDonations.toString(),
        'ether')} ETH</h2>
        <h3>Donors List</h3>
        <ul>
            {donors.map((donor, index) => (
                <li key={index}>
                    {donor.donorName} donated
        {Web3.utils.fromWei(donor.donationAmount.toString(), 'ether')} ETH
                </li>
            ))}
        </ul>
    </div>
  );
};
```

```
export default App;
```

Step 5: Run the Frontend In the client directory, run:

```
npm start
```

This will start the React app, and you can access it at http://localhost:3000

Conclusion

This project successfully implemented a Blockchain-based Charity Application that ensures transparency and trust in donations. By using Solidity smart contracts and recording transactions on the Ethereum blockchain, we eliminated the risk of data manipulation. The user-friendly frontend built with React and Web3 allows users to easily donate and view donor information. The decentralized nature of blockchain enhances accountability, making the donation process secure and transparent. This project showcases the potential of blockchain in creating a trustworthy charity system.