Electronic Health Records Maintenance Using Blockchain – System Manual

1. Introduction

The **Electronic Health Records Maintenance System** is a blockchain-based application designed to securely store, manage, and retrieve health records using Ethereum smart contracts and IPFS (InterPlanetary File System). This manual provides detailed instructions on how to set up and use the system, including how to interact with the decentralized application (DApp), upload health records, retrieve them, and manage transactions on the blockchain.

---

2. System Overview

2.1 Architecture

- **Frontend**: Built using React.js for user interaction.
- **Backend**: Smart contracts written in Solidity, deployed on the Ethereum blockchain.
- **Storage**: Files are stored on IPFS, a decentralized file storage system.
- **MetaMask**: Used for user authentication and blockchain interaction.

---

3. Installation and Setup

3.1 Prerequisites

Ensure the following tools and dependencies are installed:

- **Node.js**: Download and install from the Node.js Official Website.
- **Python**: Download and install from the Python Official Website.
- **IPFS**: Download and install from the IPFS Official Website.
- **Angular CLI**: Install Angular CLI globally:

bash

Copy code

```
npm install -g @angular/cli
```

- **Truffle Framework**: Install Truffle globally:

Copy code

```
npm install -g truffle
```

If the above command fails, try:

css

Copy code

```
npm install -g truffle@5.4.29
```

- **Git**: Download and install Git from the Git Official Website.

3.2 Setup Process

1. Open the terminal in the **Server Directory** and run the following commands:

Copy code

```
python -m pip install -r requirements.txt
python manage.py migrate
python manage.py runserver
```

2. Open **GANACHE** and import the project (blockchain workspace).
3. Open the terminal in the **Client Directory** and install dependencies:

css

Copy code

```
npm install --force
```

4. Deploy smart contracts using Truffle:

Copy code

```
truffle migrate
```

5. Start the application:

sql

Copy code

```
npm start
```

6. Open the application in your browser at:

arduino

Copy code

```
http://localhost:4200
```

---

4. User Guide

4.1 Uploading Health Records

1. **Connect MetaMask**: Click on "Connect Wallet" to link your MetaMask account to the application.
2. **Upload File**: Use the "Upload" button to select a health record file from your local machine.
3. **Confirm the Upload**: MetaMask will prompt you to confirm the transaction. Once confirmed, the record's CID will be saved on the Ethereum blockchain.
4. **Record Storage**: The file is uploaded to IPFS, and its CID (Content Identifier) is recorded on-chain.

4.2 Viewing Health Records

1. Navigate to the "My Health Records" section in the app.
2. Uploaded health records will be listed with their IPFS CIDs.
3. Click on a record’s link to view or download it.

---

5. Blockchain Interaction

5.1 Smart Contract Overview

The smart contract is responsible for:

- Storing IPFS CIDs of uploaded health records.
- Managing record ownership and permissions.
- Verifying access control and enabling secure interactions.

Contract Address: [Insert the deployed smart contract address here]

5.2 MetaMask Integration

The application uses MetaMask for:

- **Account Management**: Connecting Ethereum wallets.
- **Transaction Signing**: For uploading health records and interacting with the blockchain.

---

6. File Storage with IPFS

6.1 What is IPFS?

The **InterPlanetary File System (IPFS)** is a decentralized storage solution. Each file uploaded to IPFS is assigned a unique CID, which serves as a permanent reference to the file.

6.2 CID (Content Identifier)

Every file stored on IPFS is assigned a CID. This CID is stored on the blockchain as proof of existence and allows retrieval from IPFS by anyone with the CID.

6.3 Retrieving Files

Files can be accessed via any public IPFS gateway:

arduino

Copy code

```
https://ipfs.io/ipfs/{CID}
```

---

7. Troubleshooting

7.1 Common Issues

- **MetaMask Not Connected**: Ensure MetaMask is installed and connected to the correct Ethereum network.
- **Transaction Stuck**: Check the gas fee and ensure there is no network congestion.
- **File Not Found**: Verify the CID and ensure the file exists on the IPFS network.

---

8. Security Considerations

- **Sensitive Data**: Encrypt health records before uploading to IPFS, as they can be publicly accessed via the CID.
- **Smart Contract Risks**: Review and audit the smart contract before deploying it to the main Ethereum network.
- **MetaMask Phishing**: Use trusted devices and networks to avoid phishing attacks.

---

9. Technical Support

For further assistance, please contact:

- **Developer**: Lahari Nadendla
- **Email**: ln5k8@umsystem.edu
- **GitHub Repository**: https://github.com/Lahari529/blockchain_project.git

---

10. Appendix

10.1 Code Snippets

Smart Contract (Upload.sol):

solidity

Copy code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Upload {
  struct Access {
    address user;
    bool access; // true or false
  }
  mapping(address => string[]) private value;
  mapping(address => mapping(address => bool)) private ownership;
  mapping(address => Access[]) private accessList;
  mapping(address => mapping(address => bool)) private previousData;

  function add(address _user, string memory url) external {
    value[_user].push(url);
  }

  function allow(address user) external {
    ownership[msg.sender][user] = true;
    if (previousData[msg.sender][user]) {
      for (uint i = 0; i < accessList[msg.sender].length; i++) {
        if (accessList[msg.sender][i].user == user) {
          accessList[msg.sender][i].access = true;
        }
      }
    } else {
            accessList[msg.sender].push(Access(user, true));
            previousData[msg.sender][user] = true;
        }
    }

    function disallow(address user) public {
        ownership[msg.sender][user] = false;
        for (uint i = 0; i < accessList[msg.sender].length; i++) {
            if (accessList[msg.sender][i].user == user) {
                accessList[msg.sender][i].access = false;
            }
        }
    }

    function display(address _user) external view returns (string[] memory) {
        require(_user == msg.sender || ownership[_user][msg.sender], "Access Denied");
        return value[_user];
    }

    function shareAccess() public view returns (Access[] memory) {
        return accessList[msg.sender];
    }
}
```