Practical No - 4

Aim : Securing Transactions and Wallets

1. **Secure transaction handling in a blockchain**
2. **Setting up and securing cryptocurrency wallets**
3. **Hands-on experience with cryptographic key management**

Requirements : MEW wallet, Remix IDE, Ganache.

Theory:

Managing Keys

1. **Key Management Tools:**
   a) Use hardware wallets (like Ledger or Trezor) for storing large amounts.
   b) Use MEW or MetaMask for managing keys and making transactions.
2. **Security Best Practices:**
   a) Always double-check the recipient address before sending transactions.
   b) Enable two-factor authentication (2FA) where possible.
   c) Regularly update your wallet software.

Setting Up a Secure Wallet and Managing Keys:

Step-1:

- Go to the MEW website and create an account to go to MEW website click here.

Step-2:

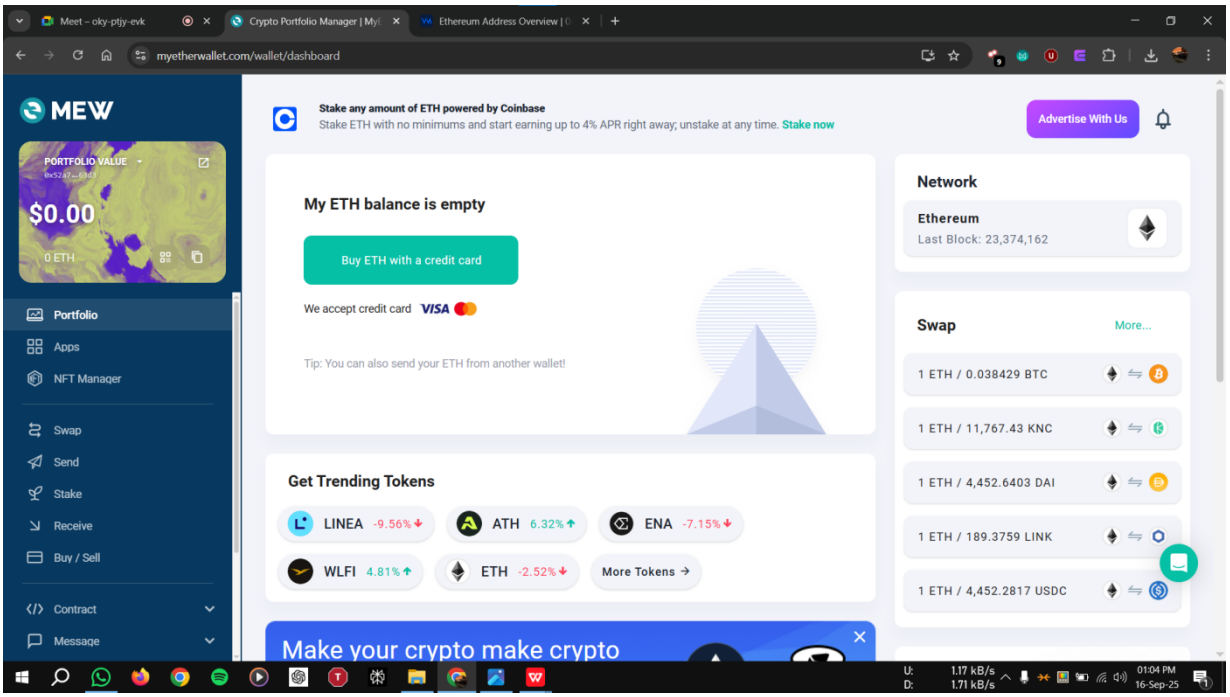
- Now click on create a new wallet, After Clicking that choose the Enkrypt Browser wallet and create an account.

Step-3:

- After clicking that the extension will be downloaded to the browser extension, after that open the extension then we need to create an account in that wallet. And then come back to the MEW website and click on access my wallet.

Step-4:

- After that click on Enkrypt then it will connect to the extension of Enkrypt and it will show the dashboard of the account like following.

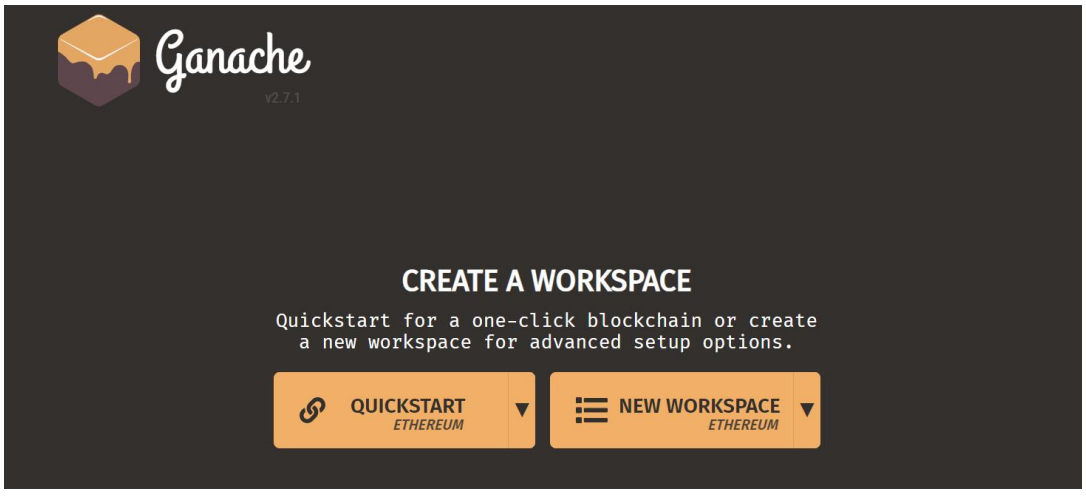


Step-5:

- While creating the account we will get the mnemonic phrase we need to keep that phrase secure and from the access my wallet we can able to download the key store file we need to keep that key store file save for further purpose.

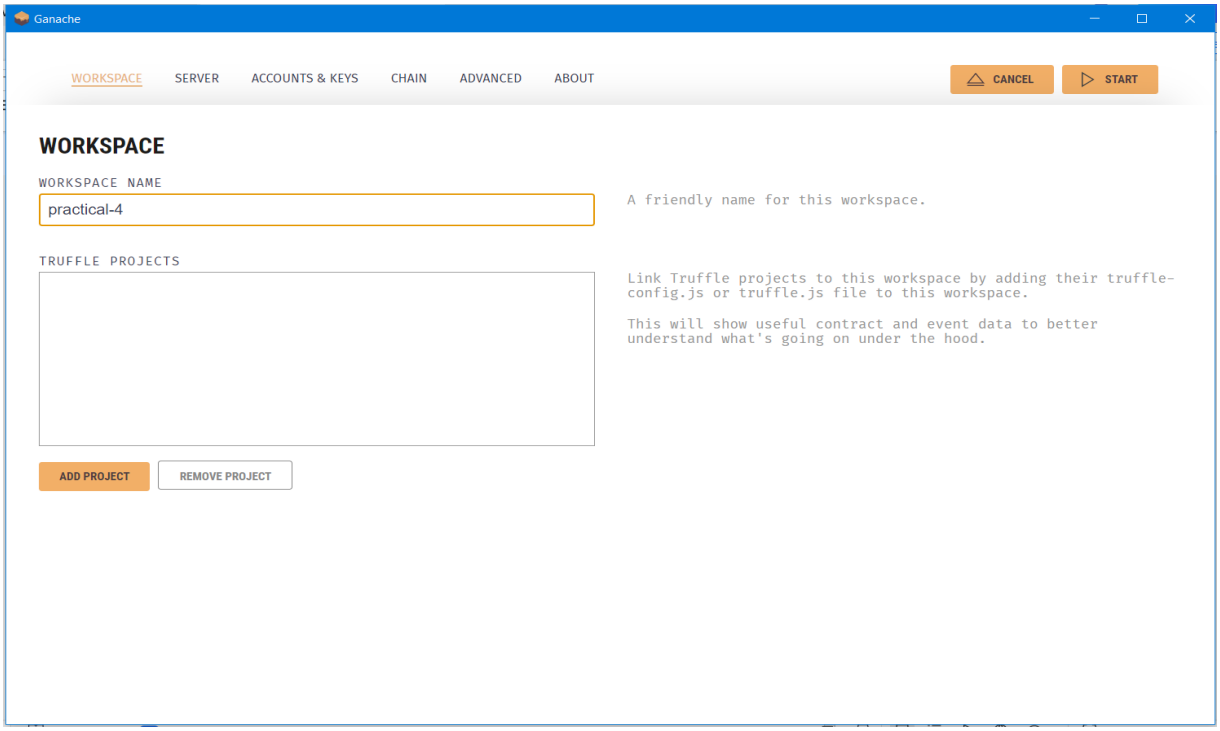
Step-6:

- Now open the ganache and click on new workspace.



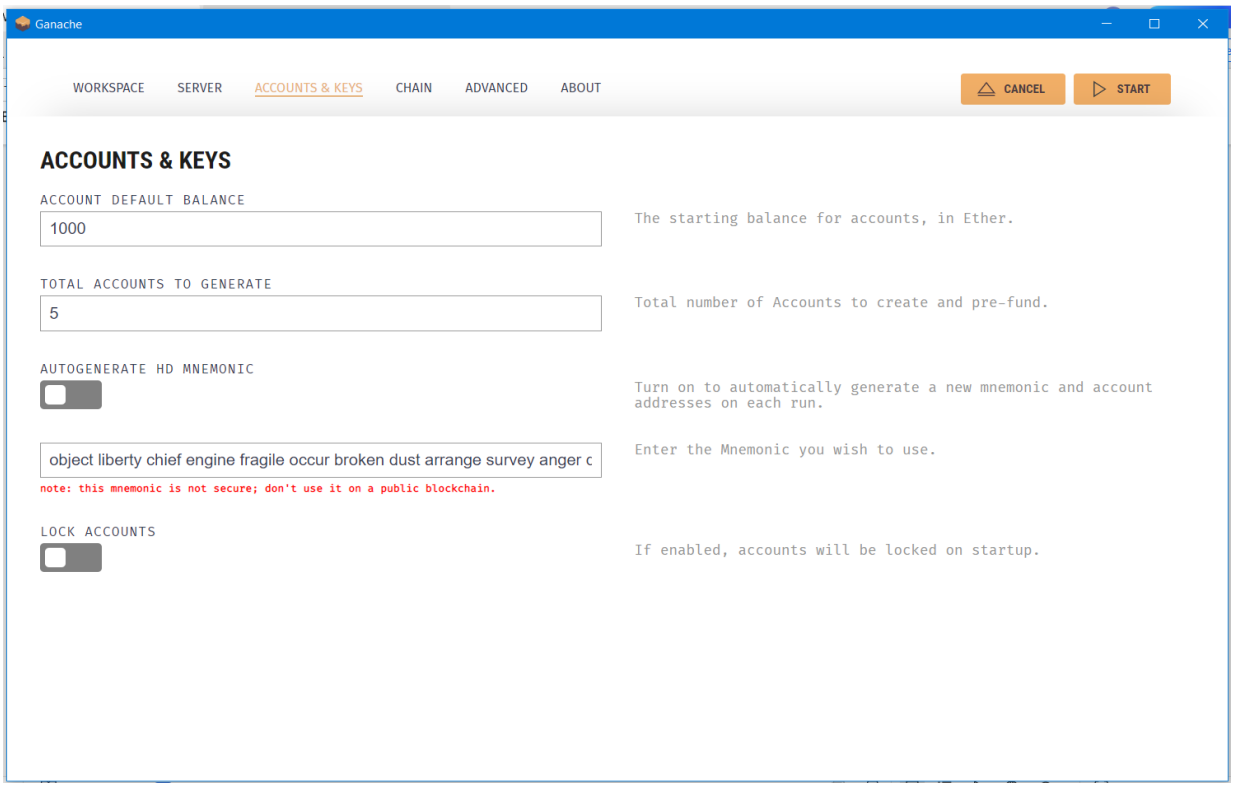
Step-7:

- After clicking on new workspace give the project name as Practical-4 and then click on the Account and Keys.



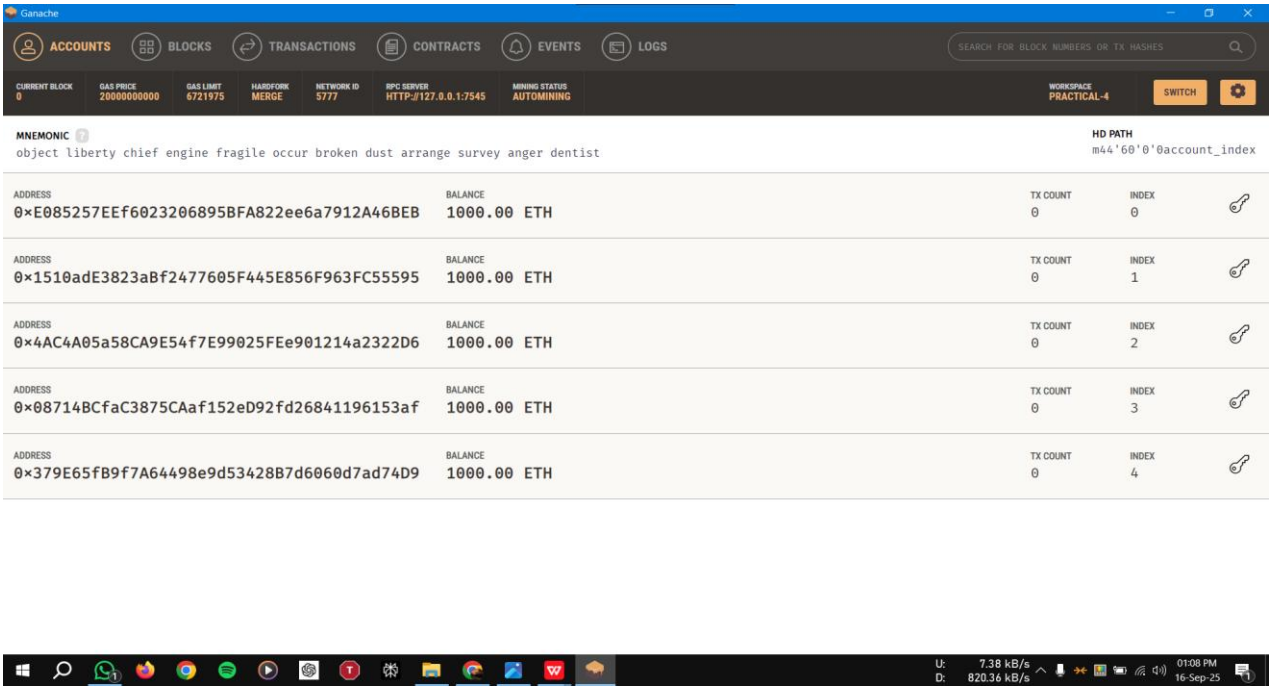
Step-8:

- After we need to go to the accounts and key option in the ganache. Over there we need to change the Accounts default balance to the 1000 and Total accounts to generate to the 5 and then hit on the start button on the right top of the ganache screen.



Step-9:

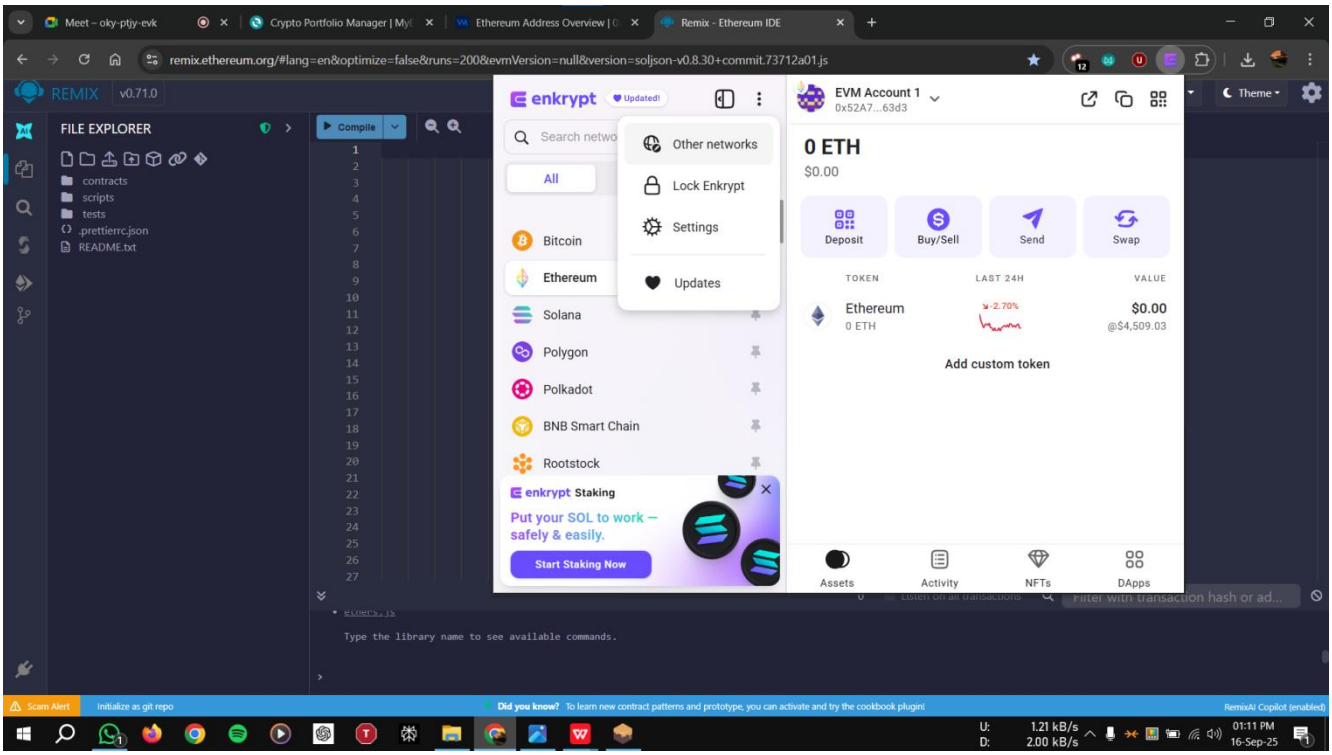
- After clicking on the start button, you can able to see the 5 accounts with the 1000 test ethers.



Connection Of MEW and Ganache

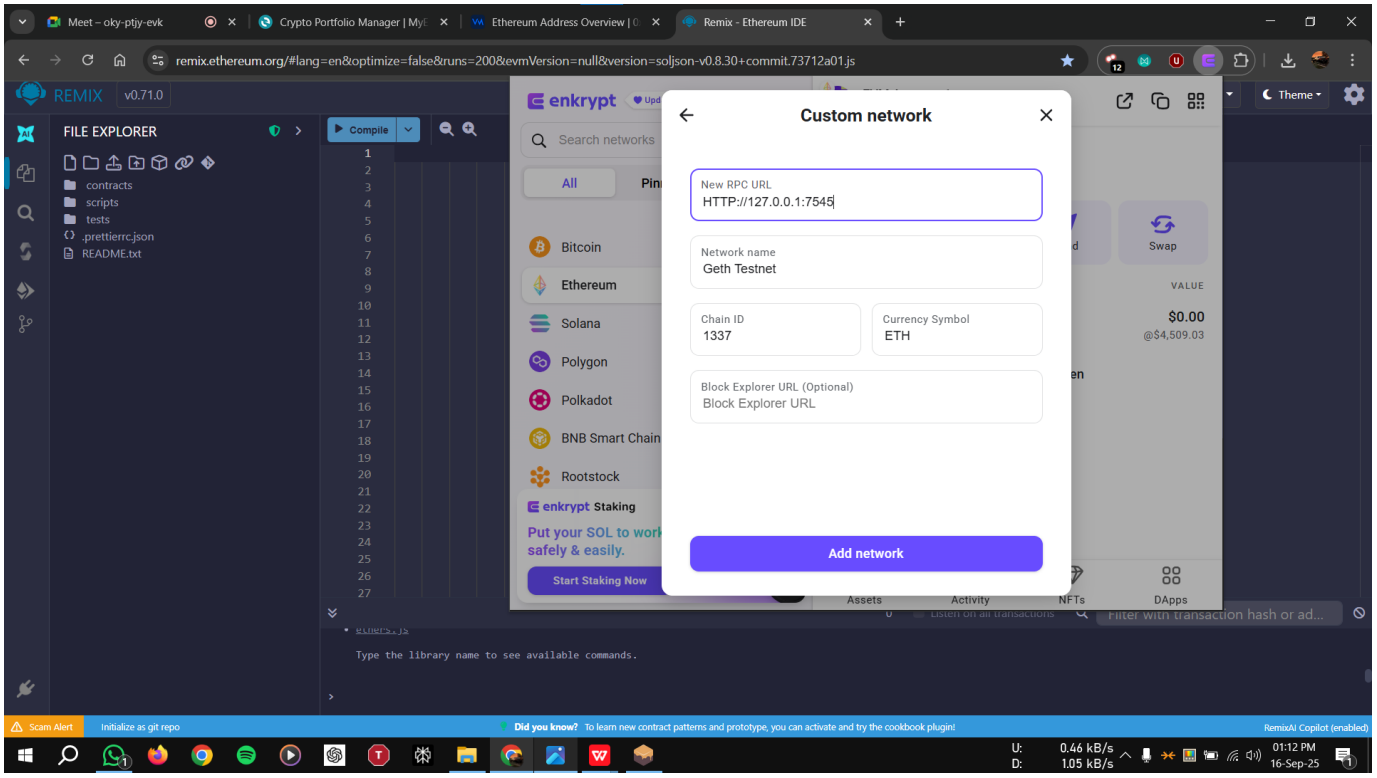
Step-10:

- First open the Encrypt in chrome and then click on other networks.



Step-11:

- Click on add Network & paste RPC url from Ganache app and you will get the network name Geth Testnet, After that click on add network to create a network.



Step-12: Now open the Remix IDE and create a file with the name prac4.sol.

Step-13: Now write insecure smart contract. As bellow given example.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract InsecureTransaction {
    event TransactionInitiated(address indexed sender, address indexed recipient, uint256 amount);
    function sendTransaction(address recipient, uint256 amount) public payable {
        // No balance check for sender
        // No validation for recipient address
        // Directly transfer amount
        payable(recipient).transfer(amount);
        emit TransactionInitiated(msg.sender, recipient, amount);
        }
    // This function allows the contract to receive Ether
    receive() external payable {}
```

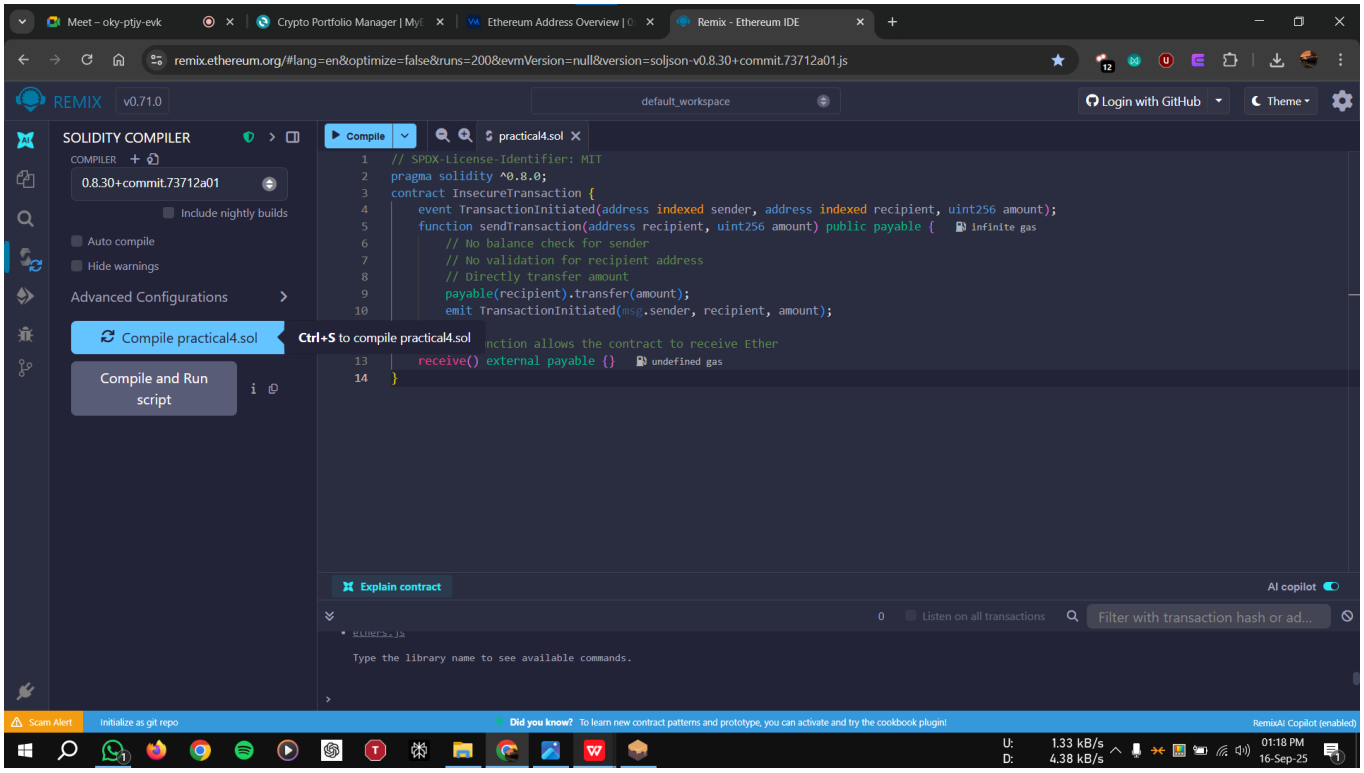
}

Step-14:

- Now we need to compile this smart contract for that purpose click on the Solidity Compiler option and then set the compiler version according to the smart contract of yours.

Step-15:

- After setting the compiler version click on compile.

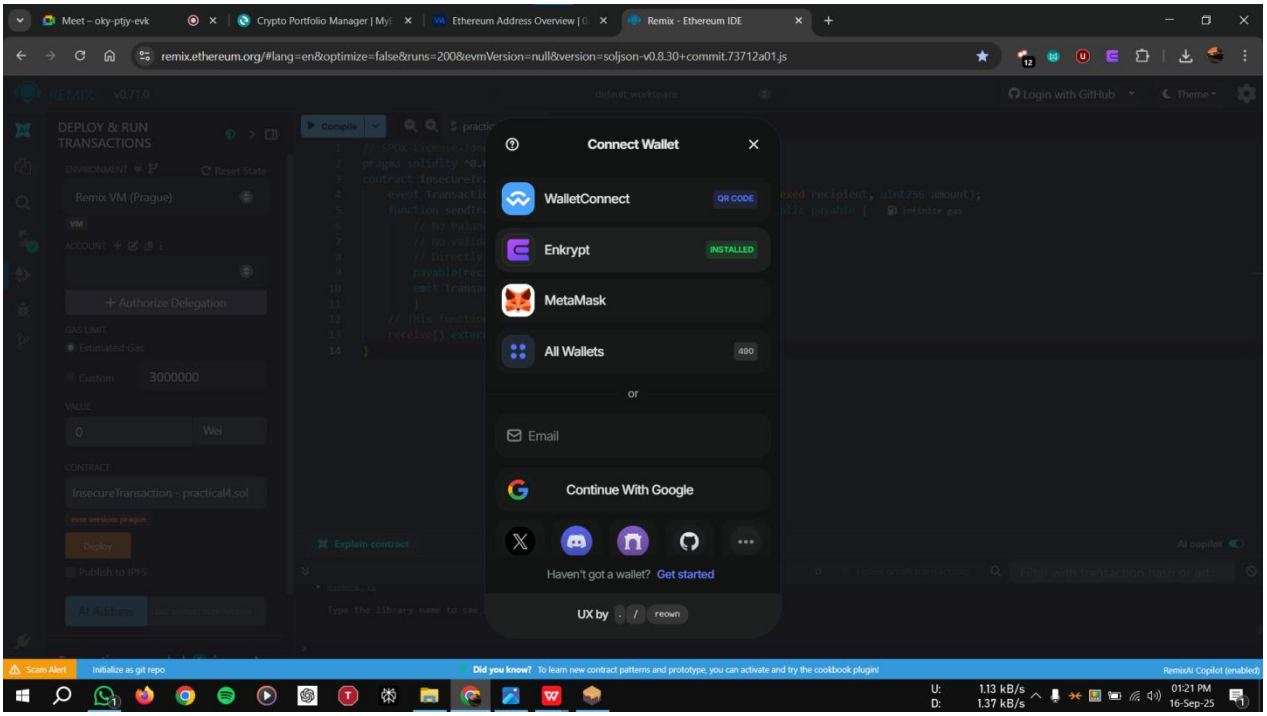


Step-16:

- After Compiling now go to the deploy and in deploy, we need to connect the Enkrypt to the Remix IDE.

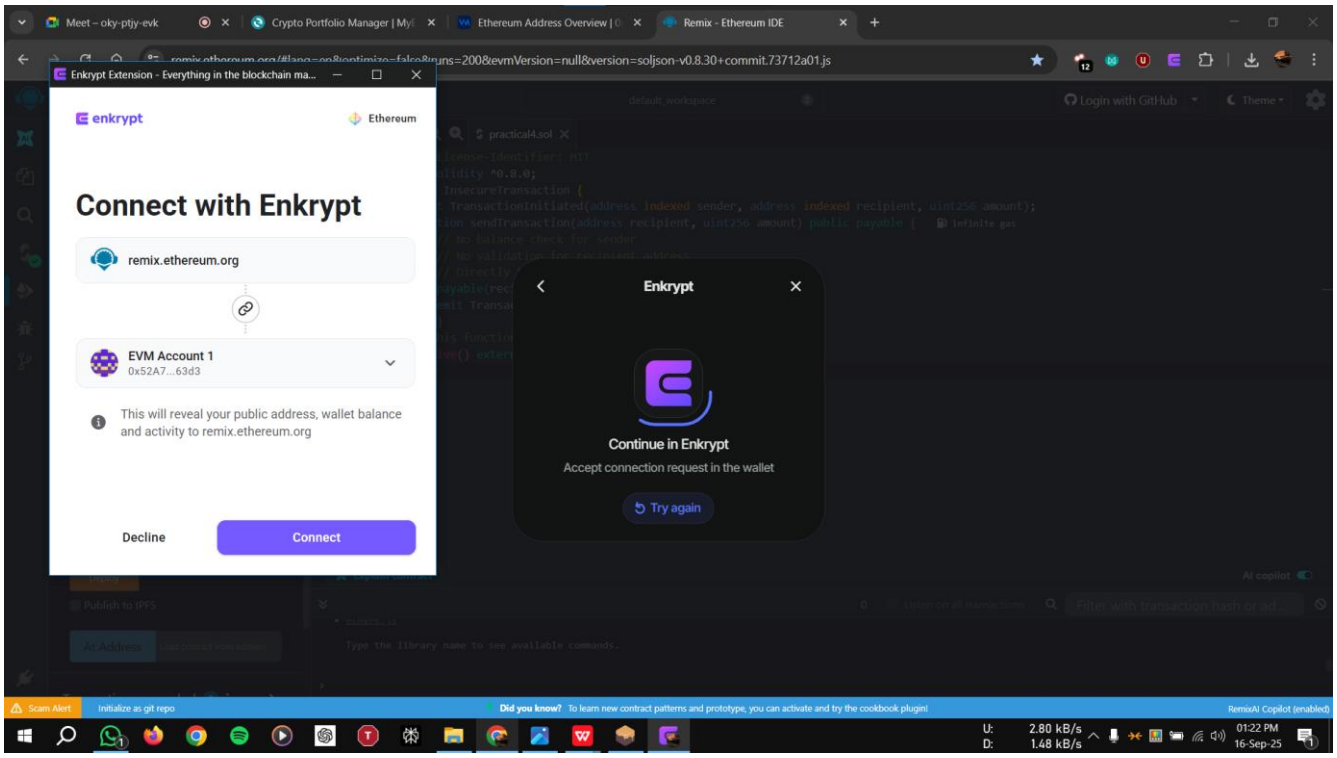
Step-17:

- To connect that Enkrypt with remix ide in deploy screen at the top we can able to see the Environment click on the dropdown and click on the Inject provider-Enkrypt.



Step-18:

- After clicking on that it will pop up one screen in that screen we need to click on the account which we had created and now click on the connect.

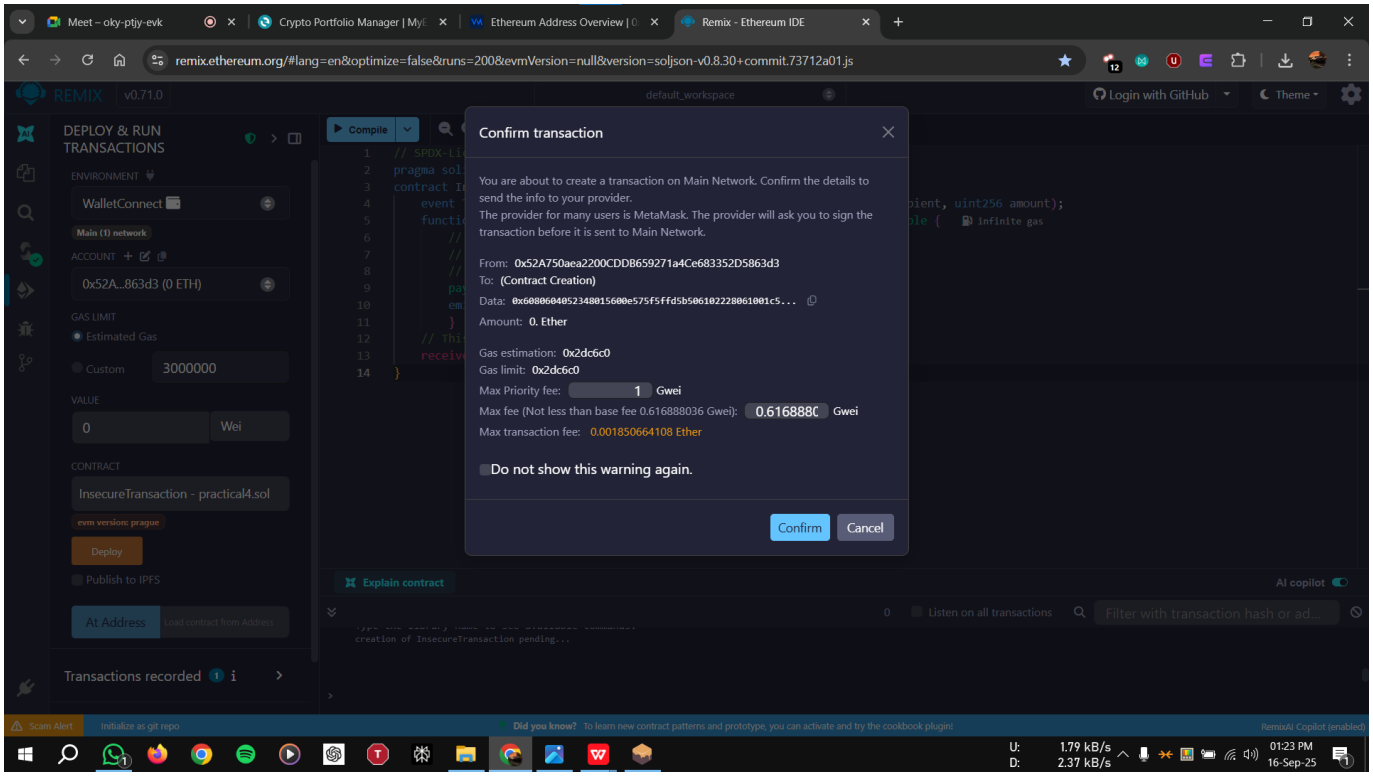


Step-19:

- After clicking that we can able to see the that account is added to it and we can able to see some ethers whatever present in that account.

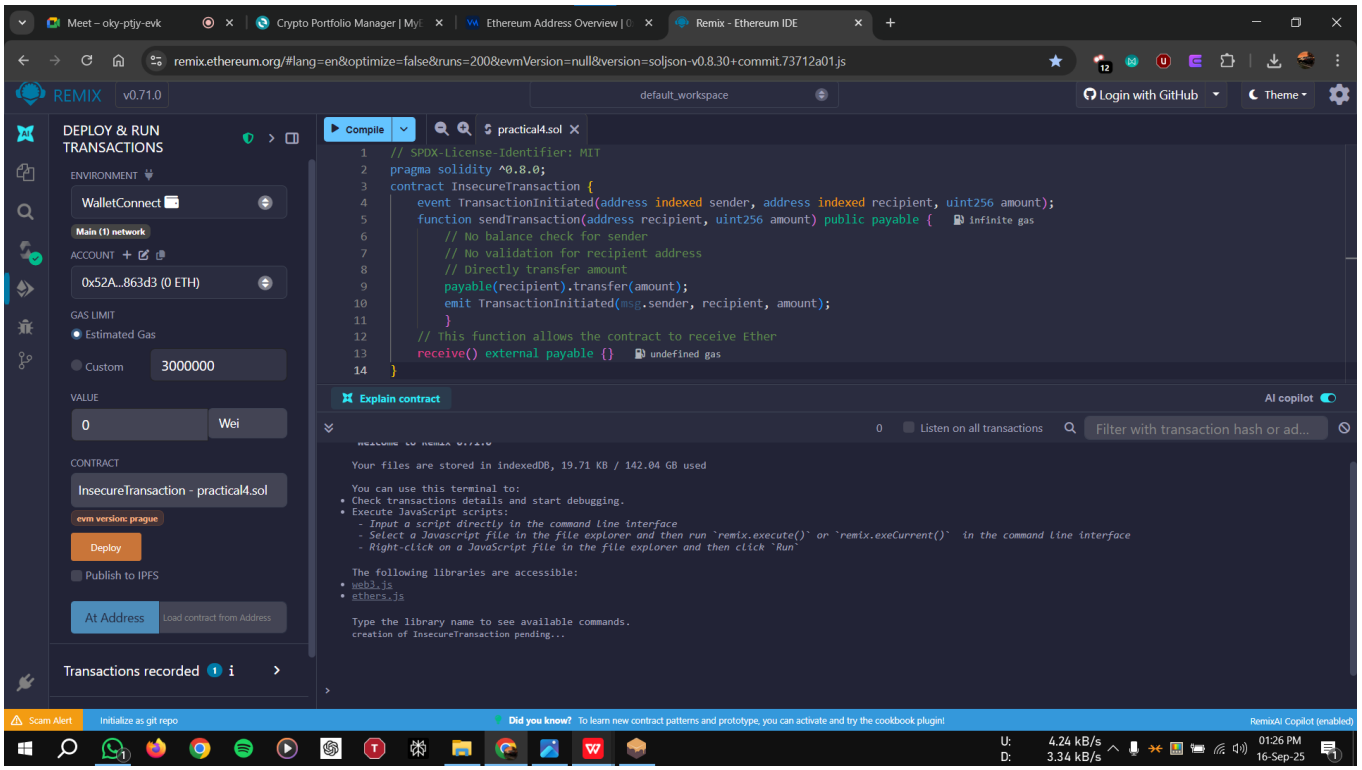
Step-20:

- Now click on the deploy button, the gas fee will generate and It will ask us to send that transaction or not.



Step-21:

- After clicking on the send the Smart contract is successfully deployed, now we can able to see the smart contract at the below of that deploy button.



Step-22:

- Now we need to give the account address to where we need to send the amount. Then will get an error because the it is an insecure smart contract.

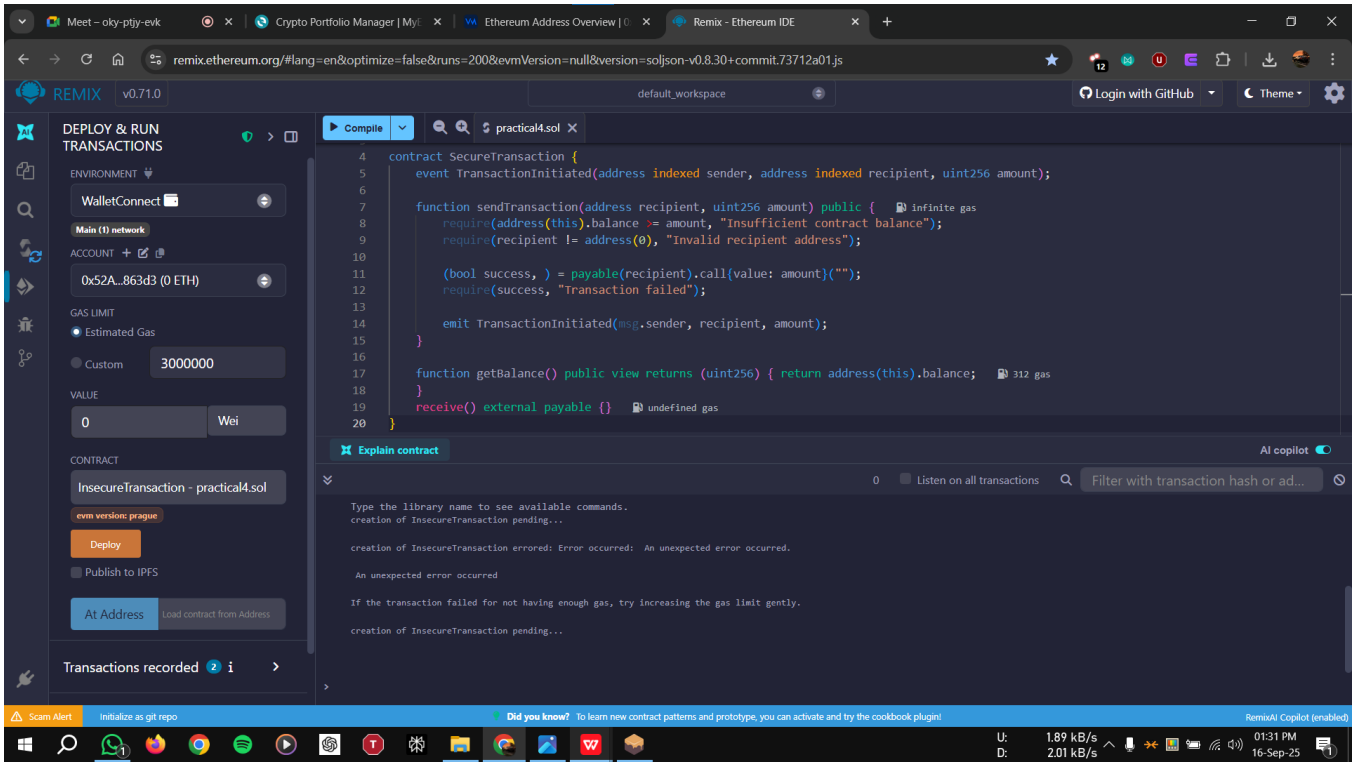
Step-23:

- Now we need to update the smart contract and we need to compile and deploy that file again the secure smart contract is as follows.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract SecureTransaction {
    event TransactionInitiated(address indexed sender, address indexed recipient, uint256 amount);
    function sendTransaction(address recipient, uint256 amount) public {
        require(address(this).balance >= amount, "Insufficient contract balance");
        require(recipient != address(0), "Invalid recipient address")
        (bool success, ) = payable(recipient).call{value: amount}("");
        require(success, "Transaction failed");
        emit TransactionInitiated(msg.sender, recipient, amount);
    }
    function getBalance() public view returns (uint256) { return address(this).balance;
    }
    receive() external payable {}
}
```

Step-24:

- Now we need to give the account number near the empty space after the send_transaction.



Conclusion:

Hence, we concluded the working of a secure smart contract as well as learnt how to setup a secure wallet with key management.

Practical No - 5

Aim: Introduction to Web 3.0 & Setting Up Basic DApp Environment

- Understand the evolution from Web 2.0 to Web 3.0.
- Explore decentralized applications (DApps).
- Set up a basic DApp environment using Ganache, Truffle, and MetaMask.

Requirements:

- Ganache (Ethereum Blockchain Simulator)
- Node.js
- Truffle Framework
- Visual Studio Code (or any text editor)
- MetaMask Wallet

Theory:

Understanding the Evolution from Web 2.0 to Web 3.0:

Web 2.0 vs Web 3.0

- **Web 2.0:**

Refers to the second generation of the World Wide Web.

Focuses on user-generated content, usability, and interoperability.

Emphasizes social networking sites, blogs, wikis, and video-sharing platforms where users interact and collaborate on content.

Example of Web 2.0:

Facebook: Allows users to create profiles, share content, and interact with others on a centralized platform controlled by Facebook.

- **Web 3.0:**

Known as the decentralized web.

Builds on Web 2.0 with a focus on decentralization, blockchain technology, and token-based economics.

Aims to give users more control over their data and digital identities.

Example of Web 3.0:

Ethereum-based social media platforms like Minds: Users own their data, earn tokens for contributions, and interact on a decentralized network.

Exploring Decentralized Applications (DApps)

What are DApps?

Applications that run on a blockchain or peer-to-peer network instead of centralized servers.

They are open-source, operate autonomously, and use smart contracts to enforce rules and interactions without a central authority.

Example of a DApp:

Uniswap: A decentralized exchange on the Ethereum blockchain that allows users to trade cryptocurrencies without a central intermediary.

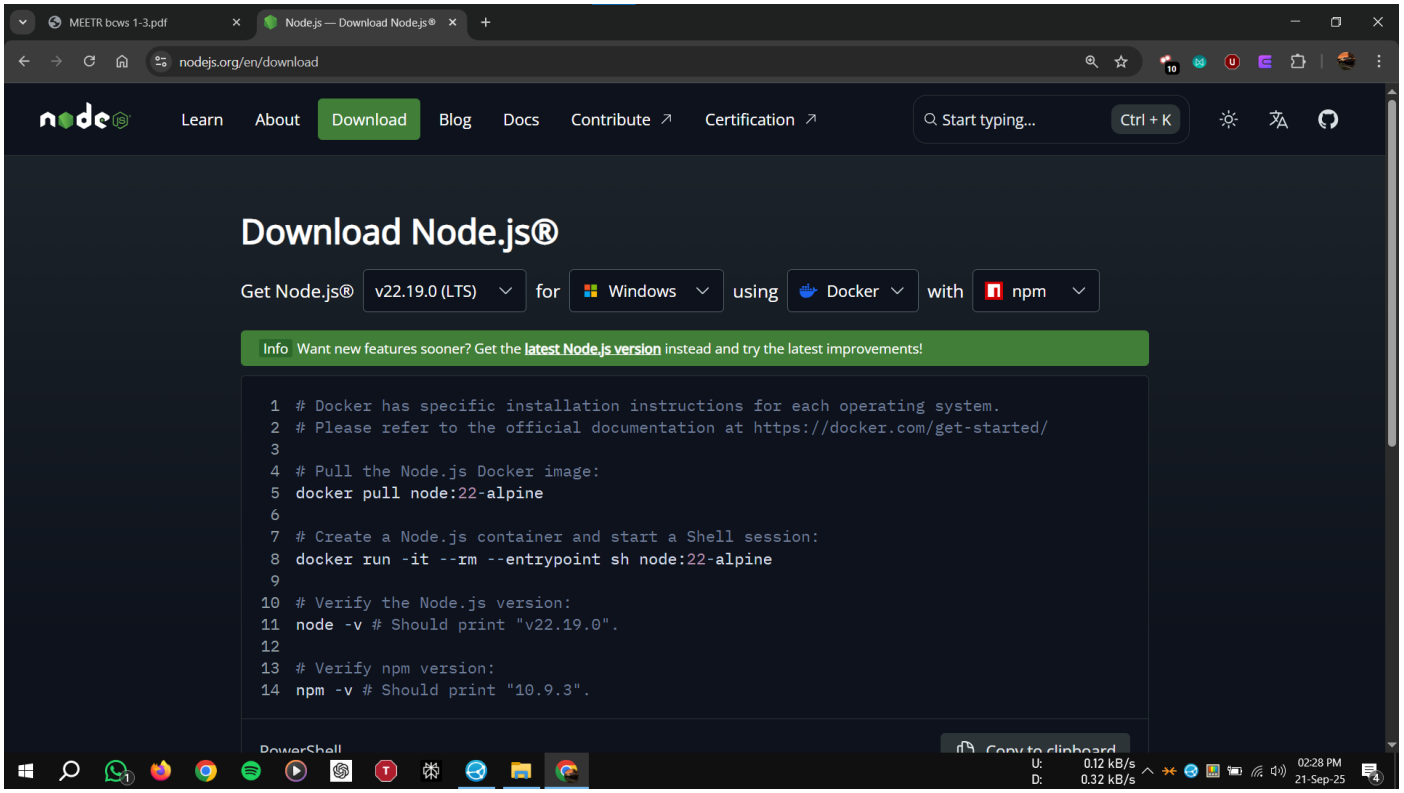
Procedure: Setting Up a Basic DApp Environment

Step 1: Install Node.js

Download Node.js from the official website: https://nodejs.org

Choose the LTS (Long Term Support) version for better stability.

Run the installer and complete the installation.



Step 2: Verify Node.js Installation

Open Command Prompt or Terminal.

Run the command:

node --version

It should display the installed Node.js version.

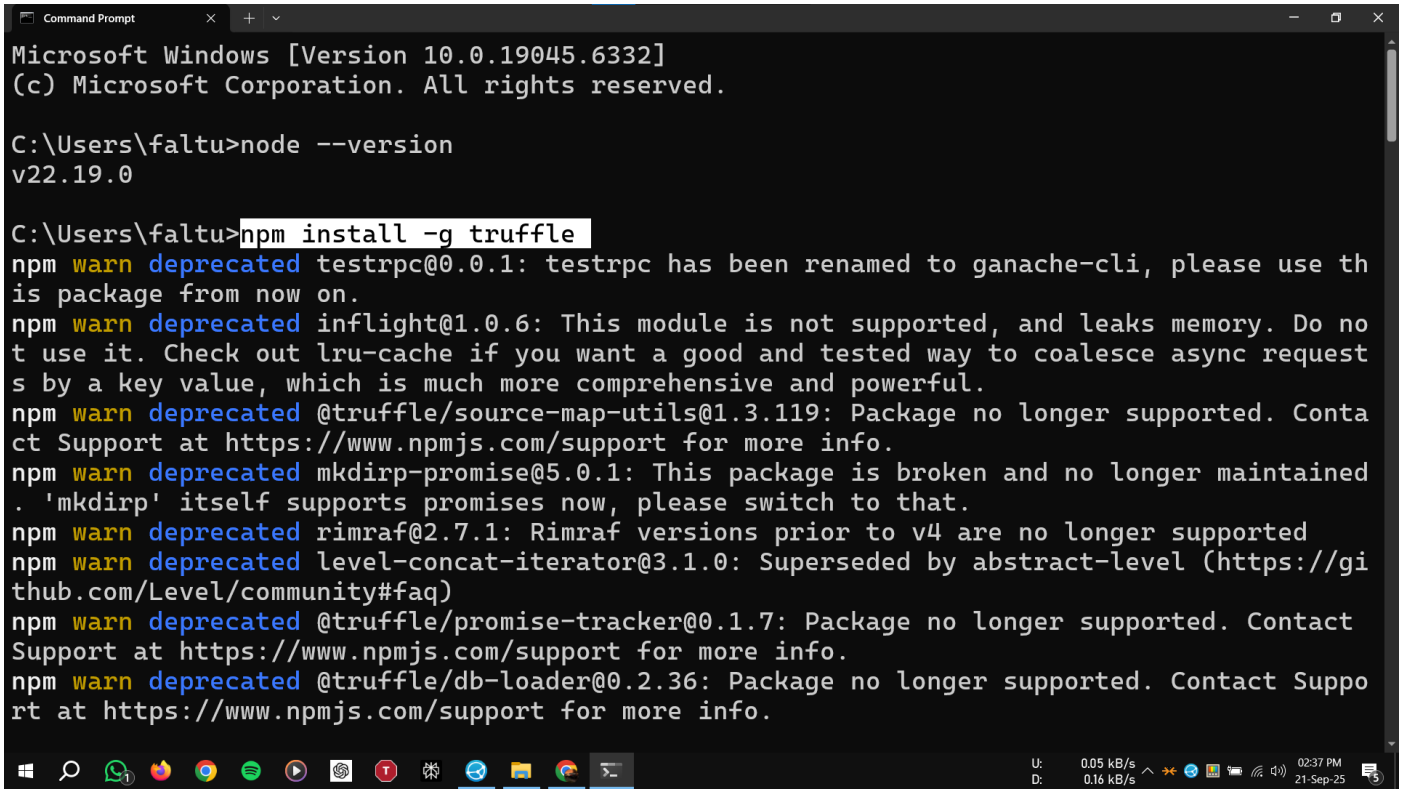
```
C:\Users\faltu>node --version
v22.19.0

C:\Users\faltu>
```

Step 3: Install Truffle Framework

Run the following command in terminal:

npm install -g truffle

This installs Truffle globally.



Step 4: Install and Open Ganache

Download Ganache from: https://trufflesuite.com/ganache

Install and open Ganache.

Click New Workspace.

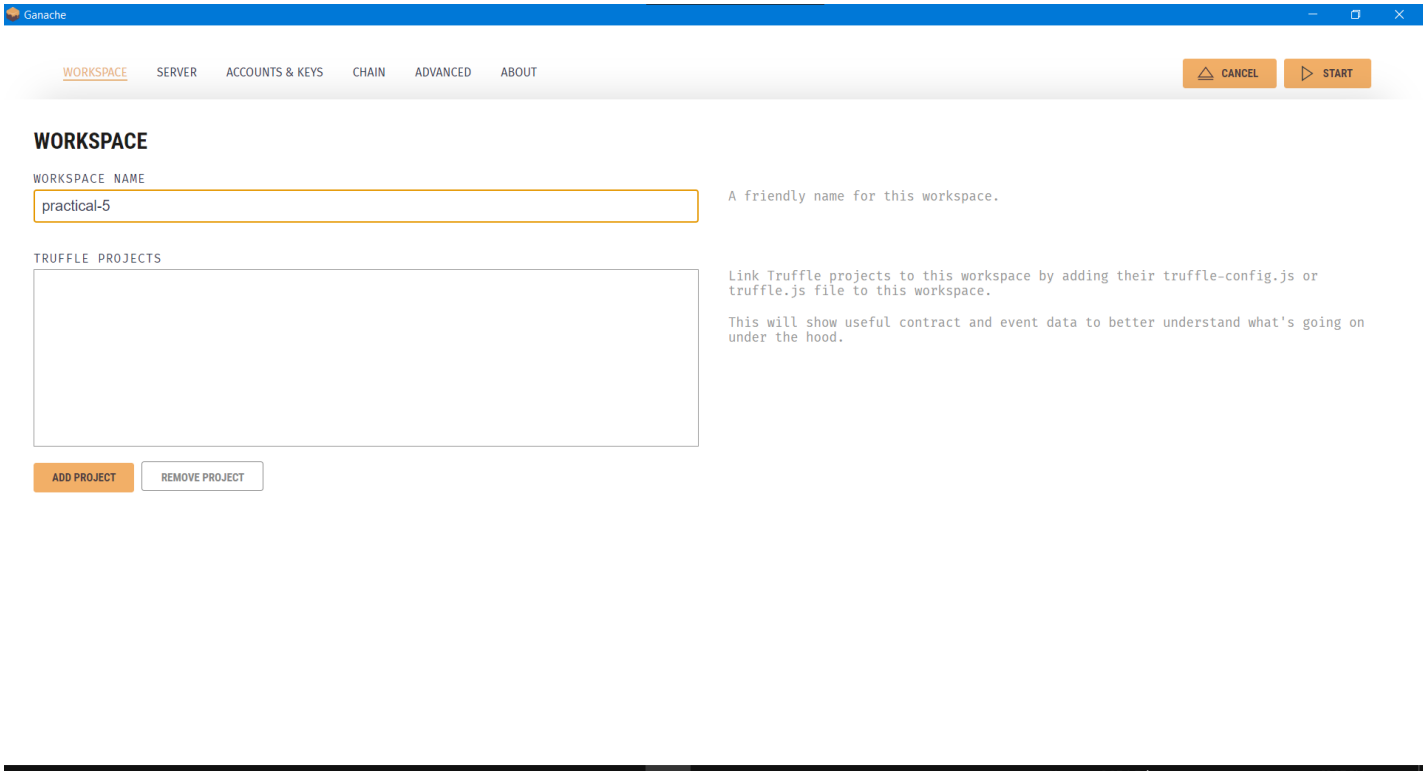
Rename project to practical-5.

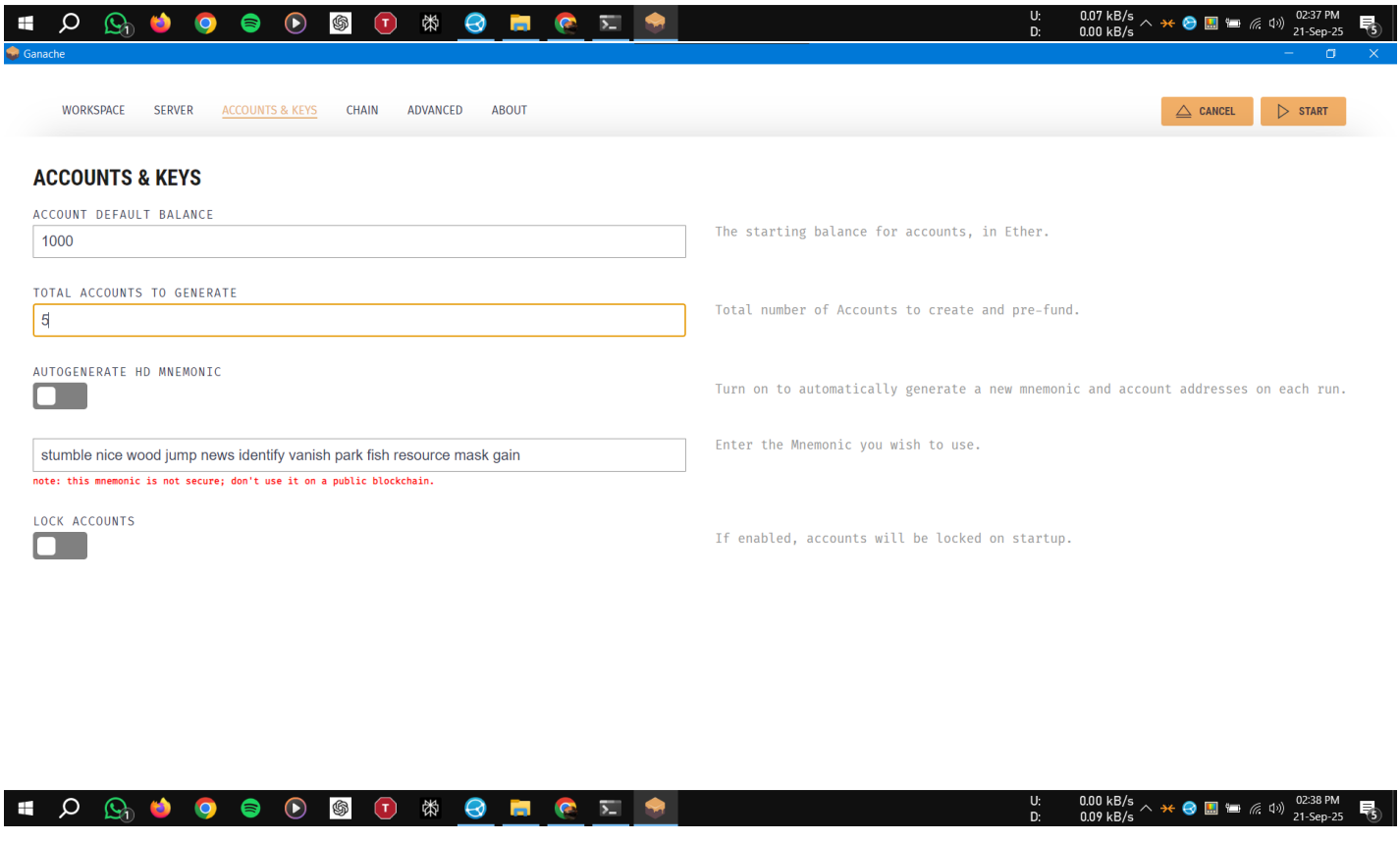
Go to Accounts & Keys.

Set number of accounts to 5.

Set balance for each account to 1000 ETH.

Click Start to run the local blockchain.





Step 5: Create Truffle Project

Open Visual Studio Code (VSCode).

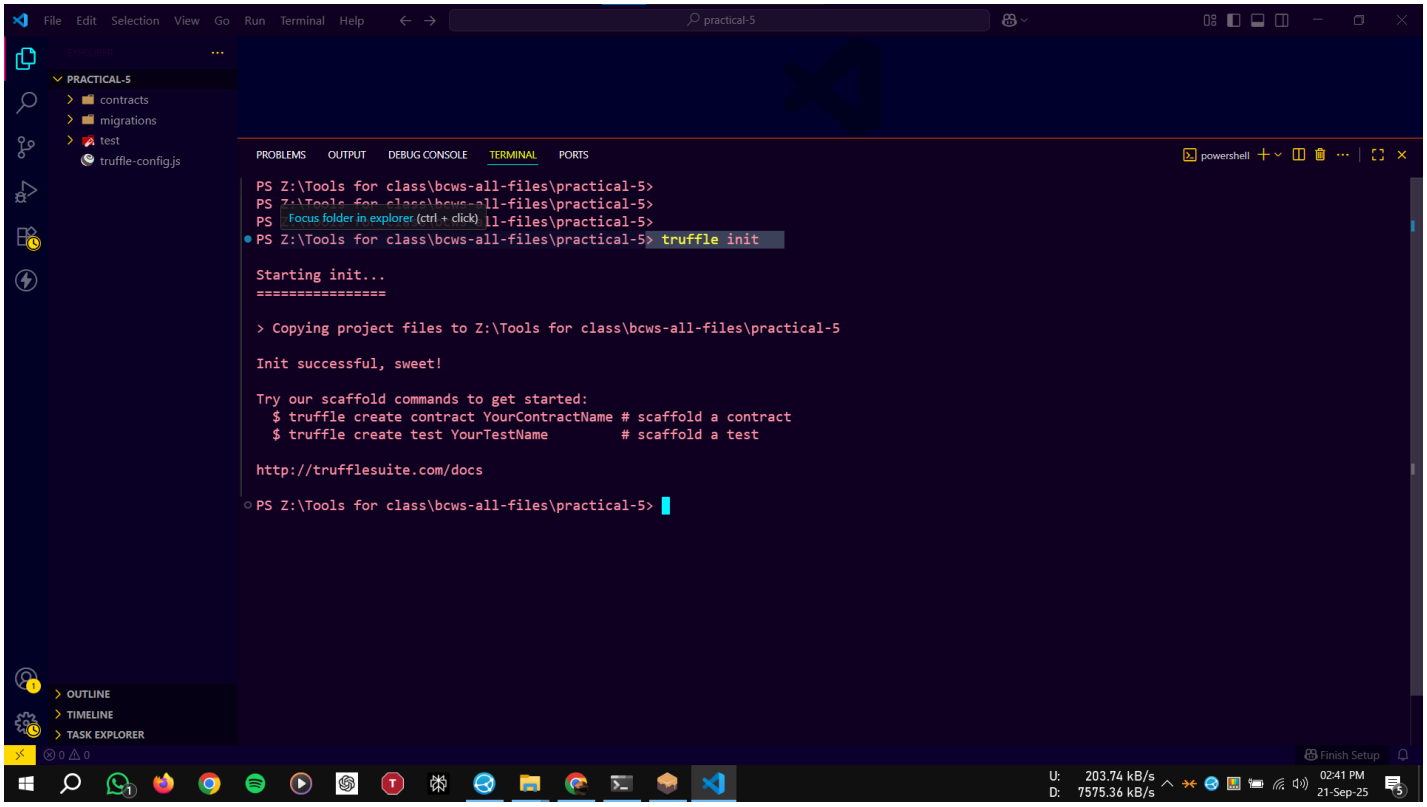
Create a new folder (e.g., practical-5).

Open terminal inside VSCode at the project folder.

Initialize a Truffle project by running:

truffle init

This will create default folders and files.



Step 6: Configure Truffle to Connect with Ganache

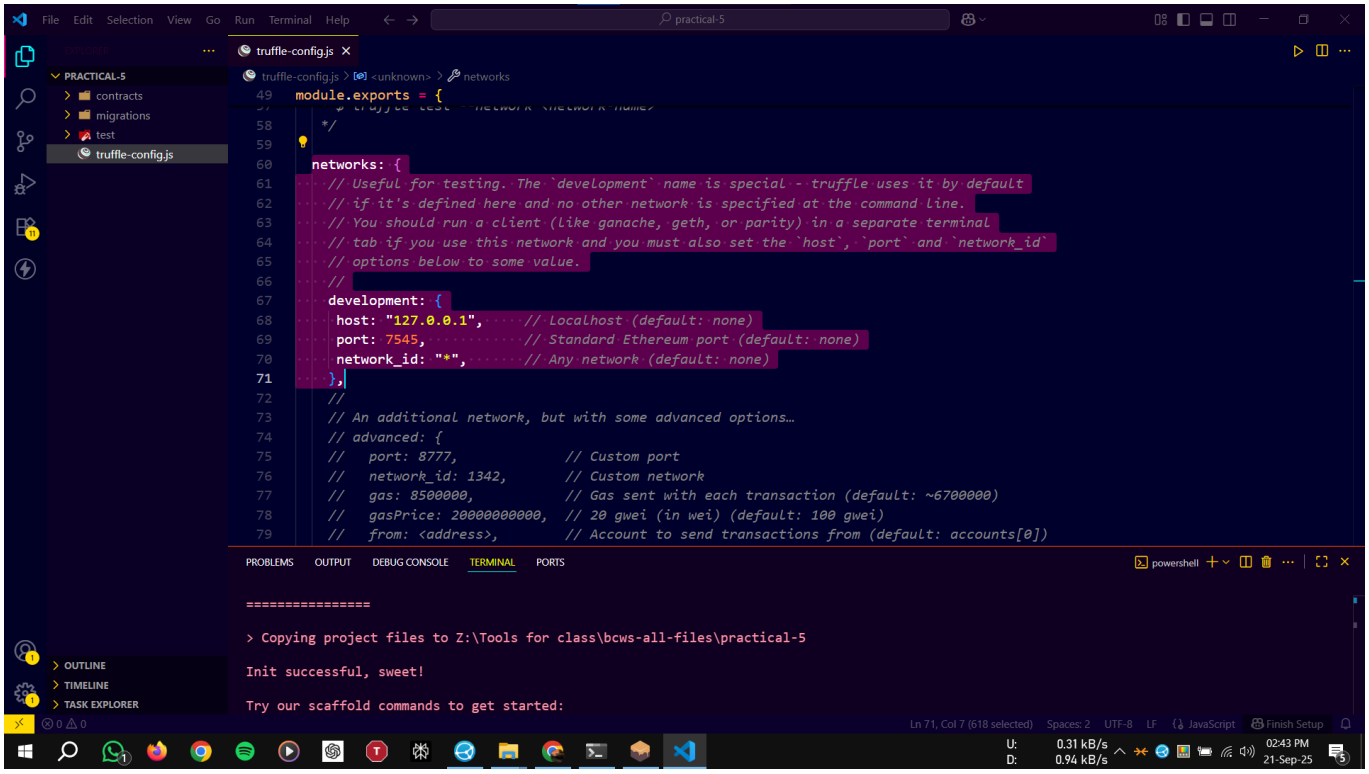
Open the truffle-config.js file.

Locate the networks section.

Uncomment and add this configuration:

```
networks: {
    development: {
        host: "127.0.0.1",
        port: 7545,                 // Ganache default port
        network_id: "*",          // Match any network id
    },
},
```

Save the file.



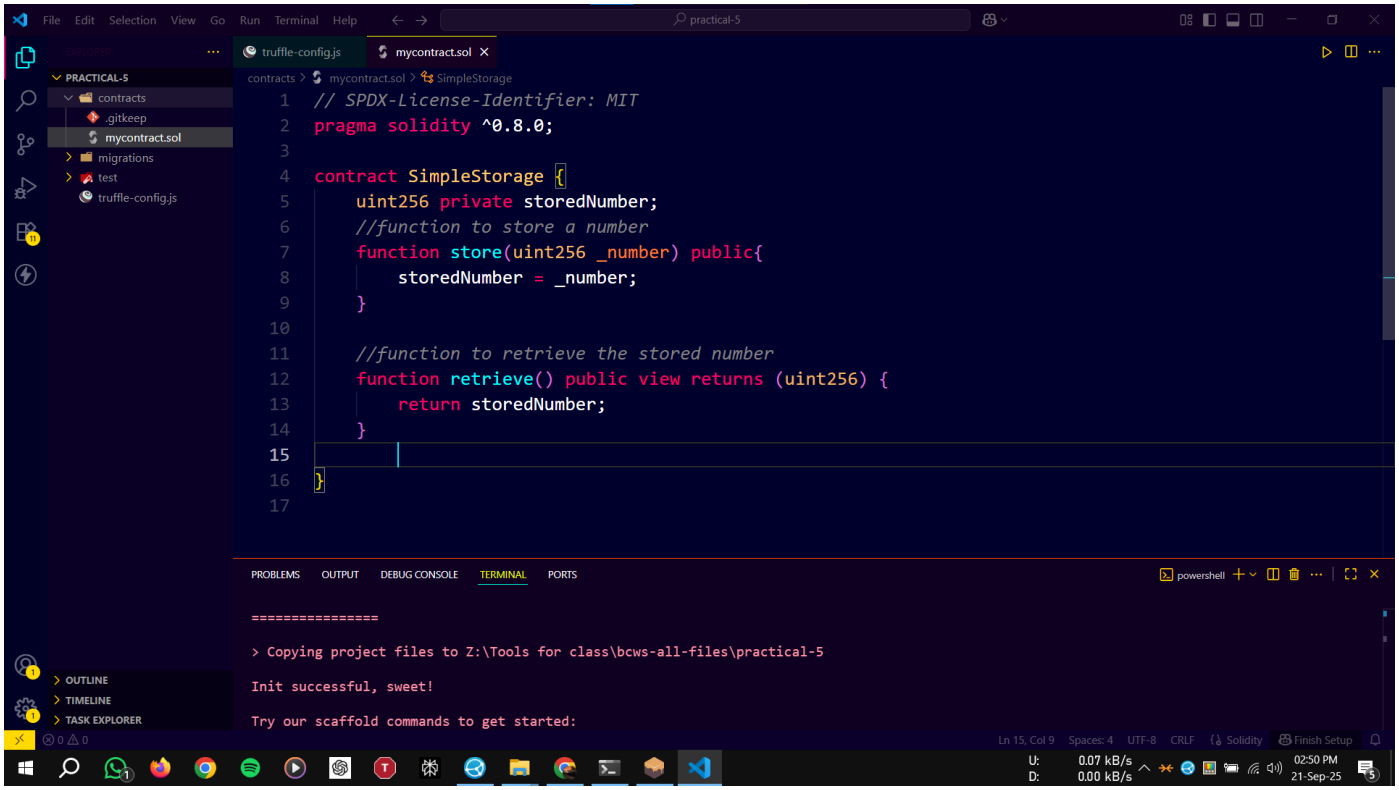
Step 7: Create Smart Contract

Inside the contracts folder, create a new file named MyContract.sol.

Write a simple contract, for example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract SimpleStorage {
    uint256 private storedNumber;
    //function to store a number
    function store(uint256 _number) public{
        storedNumber = _number;
    }
    //function to retrieve the stored number
    function retrieve() public view returns (uint256) {
        return storedNumber;
    }
}
```

Save the contract.



Step 8: Write Migration Script

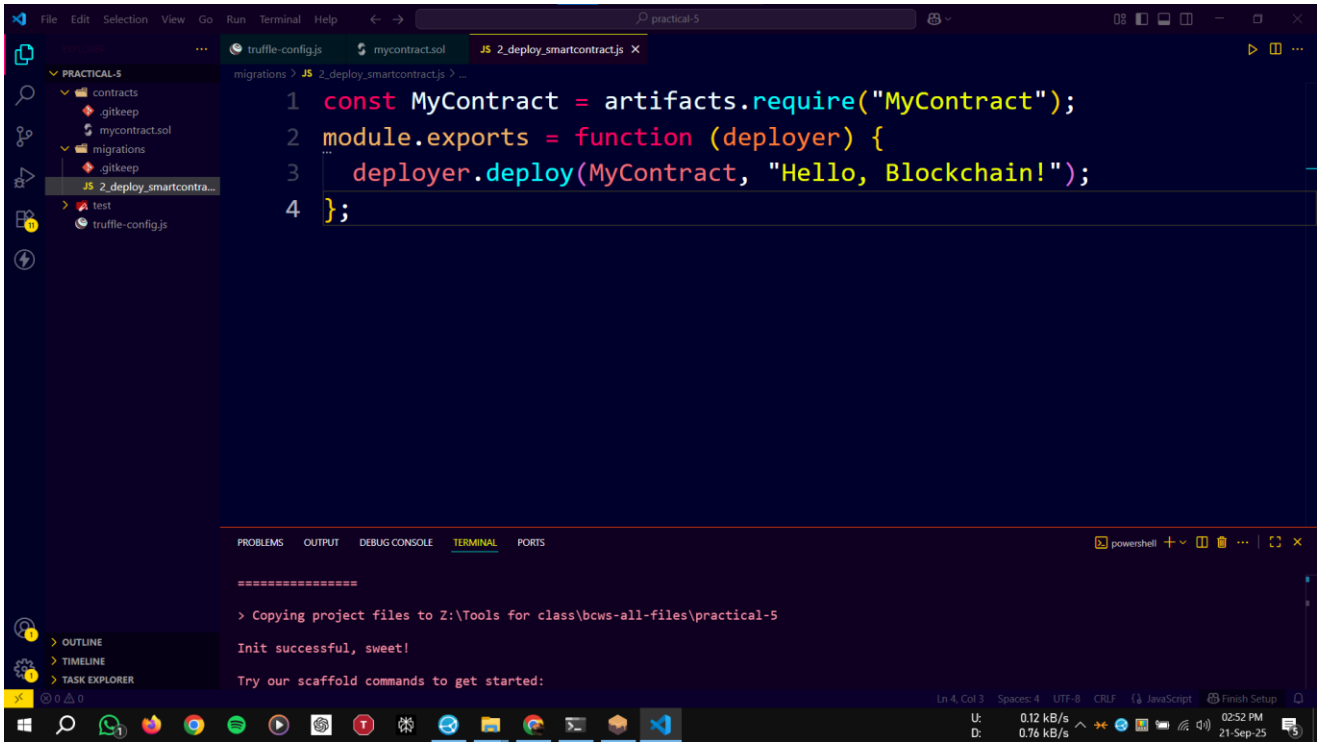
Open the migrations folder.

Create a new file named *2_deploy_smartcontract.js.*

Add the following code:

```
const MyContract = artifacts.require("MyContract");

module.exports = function (deployer) {
    deployer.deploy(MyContract, "Hello, Blockchain!");
};
```

Save the file.



Step 9: Compile and Deploy Contract

In terminal, run:

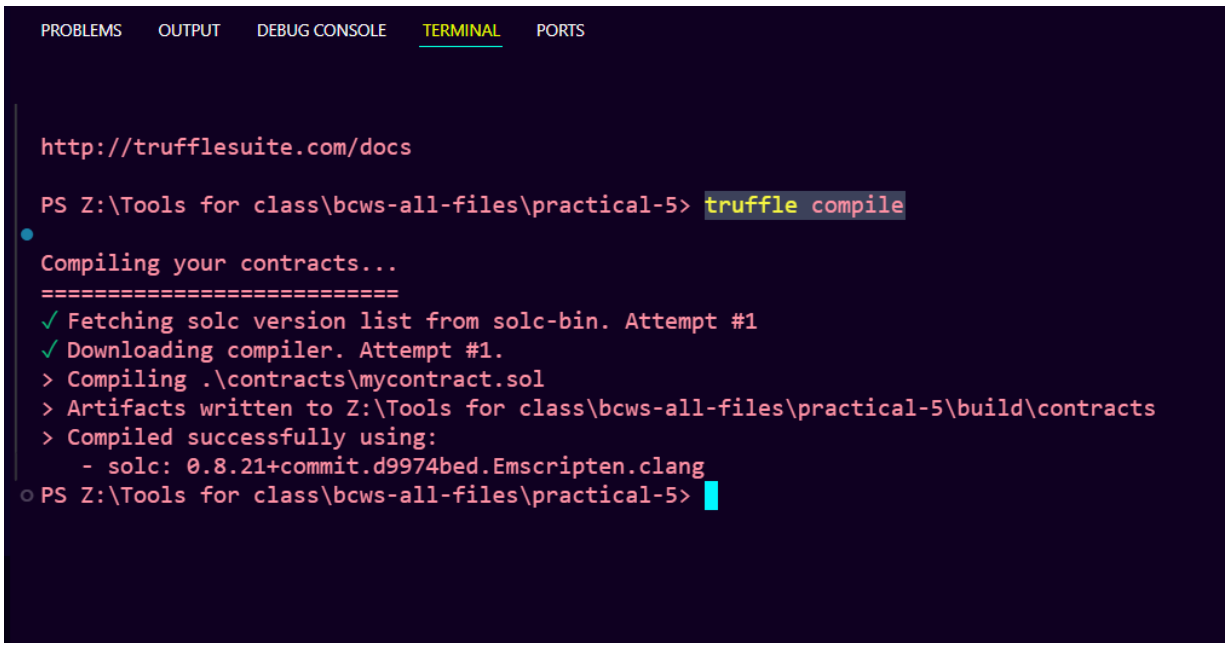
truffle compile

This compiles your smart contracts.

Then run:

truffle migrate --network development

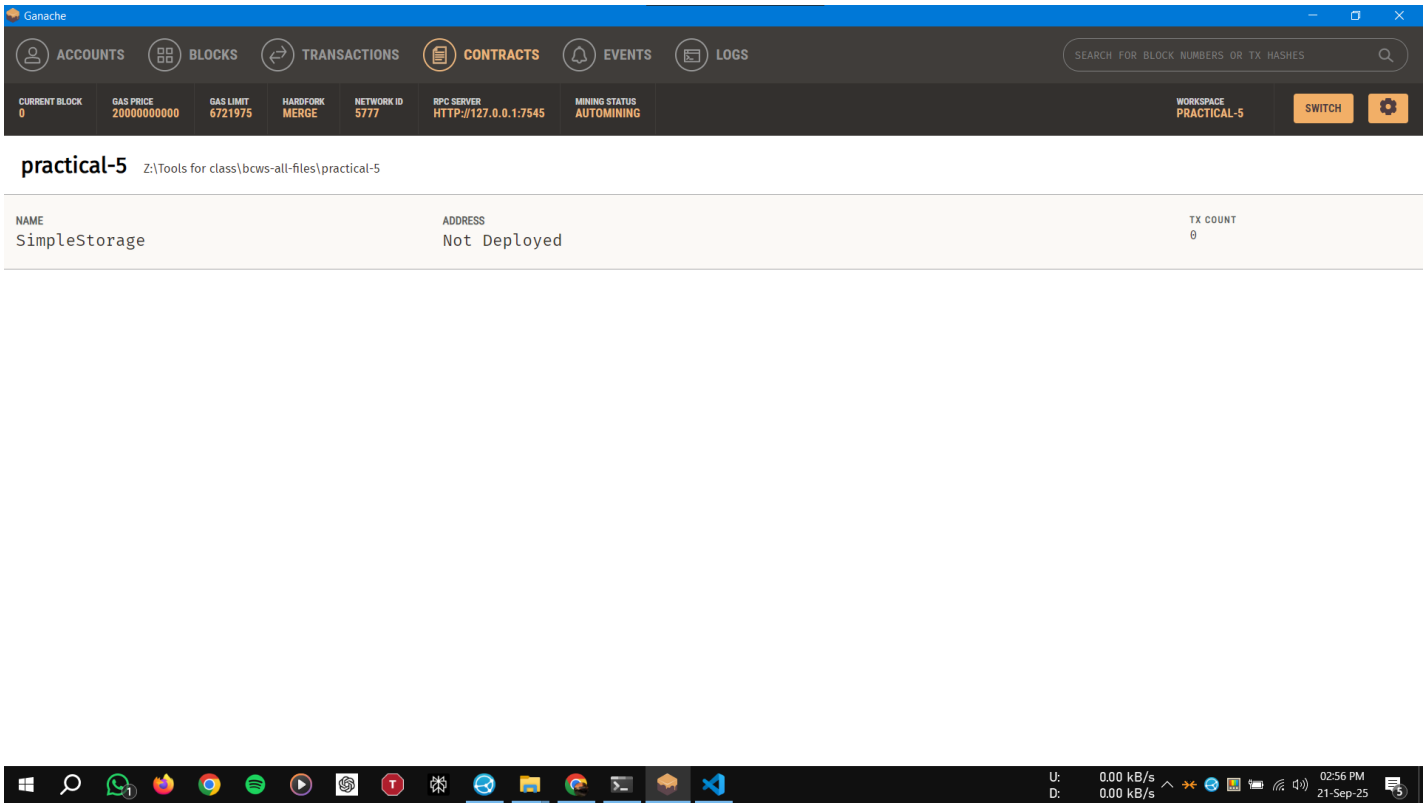
This deploys the contract to Ganache blockchain.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    powershell

PS Z:\Tools for class\bcws-all-files\practical-5> truffle migrate --network development
This version of µWS is not compatible with your Node.js build:

Error: Cannot find module '../binaries/uws_win32_x64_127.node'
Require stack:
- C:\Users\faltu\AppData\Roaming\npm\node_modules\truffle\node_modules\ganache\node_modules\@trufflesuite\uws-js-unofficial\src\uws.js
- Open file in editor (ctrl + click) \Roaming\npm\node_modules\truffle\node_modules\ganache\dist\node\core.js
- C:\Users\faltu\AppData\Roaming\npm\node_modules\truffle\build\migrate.bundled.js
- C:\Users\faltu\AppData\Roaming\npm\node_modules\truffle\node_modules\original-require\index.js
- C:\Users\faltu\AppData\Roaming\npm\node_modules\truffle\build\cli.bundled.js
Falling back to a NodeJS implementation; performance may be degraded.


Compiling your contracts...
===========================
> Compiling .\contracts\mycontract.sol
> Artifacts written to Z:\Tools for class\bcws-all-files\practical-5\build\contracts
> Compiled successfully using:
```

Step 10: Verify Deployment on Ganache

Open Ganache GUI.



Conclusion:

We learned about Web 3.0 and how it differs from Web 2.0.

Set up a local blockchain environment using Ganache.

Created, compiled, and deployed a smart contract using Truffle.

Connected your DApp environment to Ganache to simulate blockchain transactions.

Practical No - 6

Aim: Developing and Deploying a Simple DApp.

1. Create a decentralized application (DApp) using a blockchain platform.

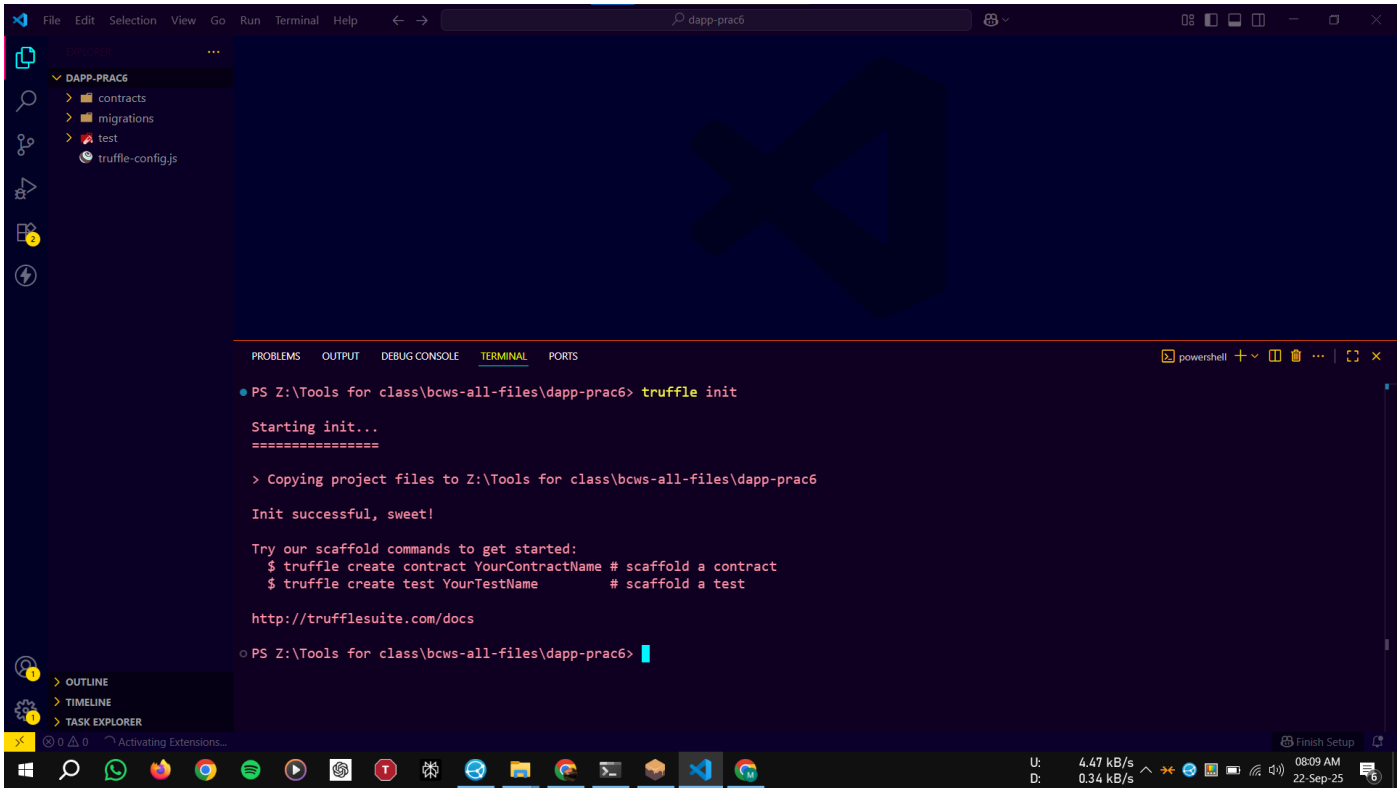
2. Deploy the DApp on a test network.

3. Interact with the DApp through a web interface.

Steps to create simple Dapp

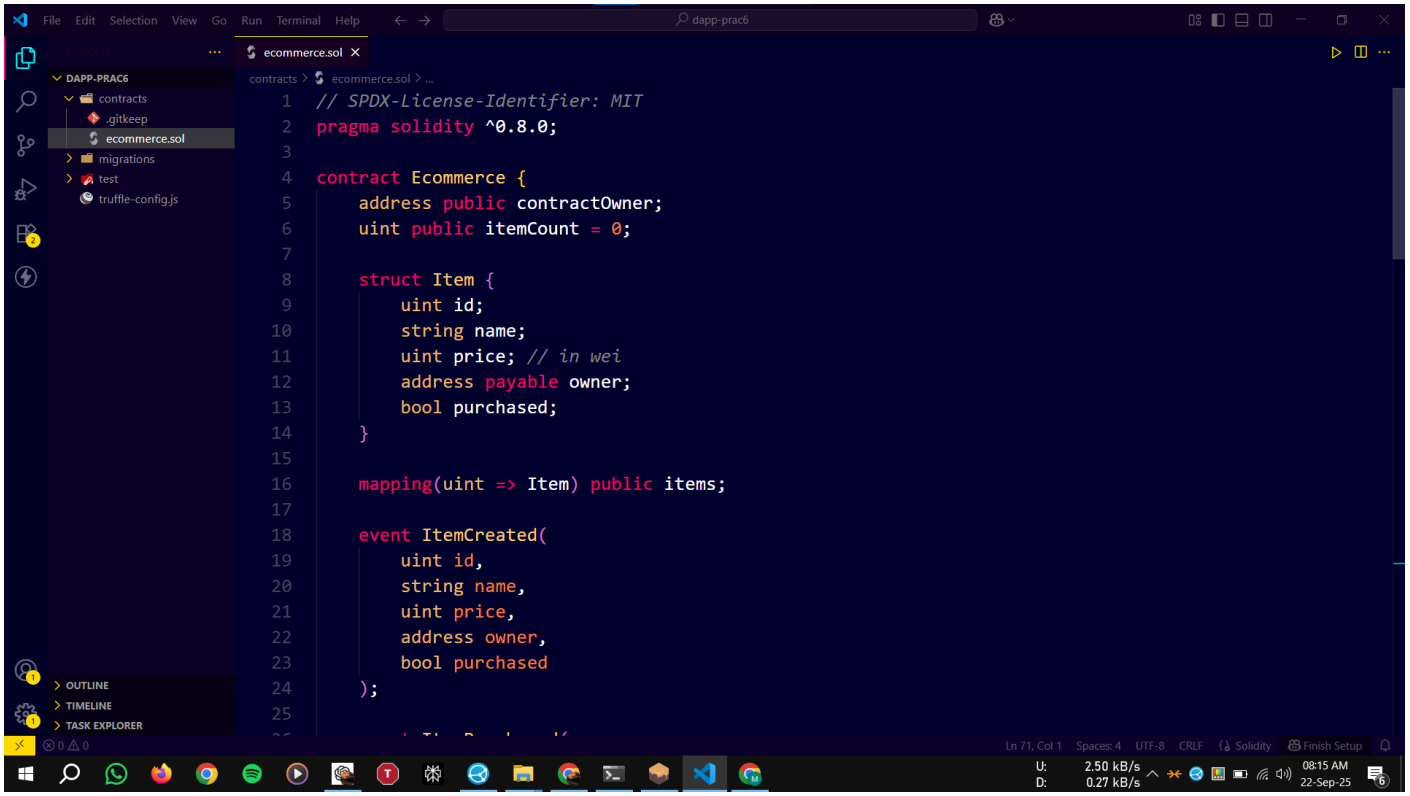
Step-1

Open new folder in vscode and first initialize truffle project into this directory by giving the following command "*truffle init*"



Step-2

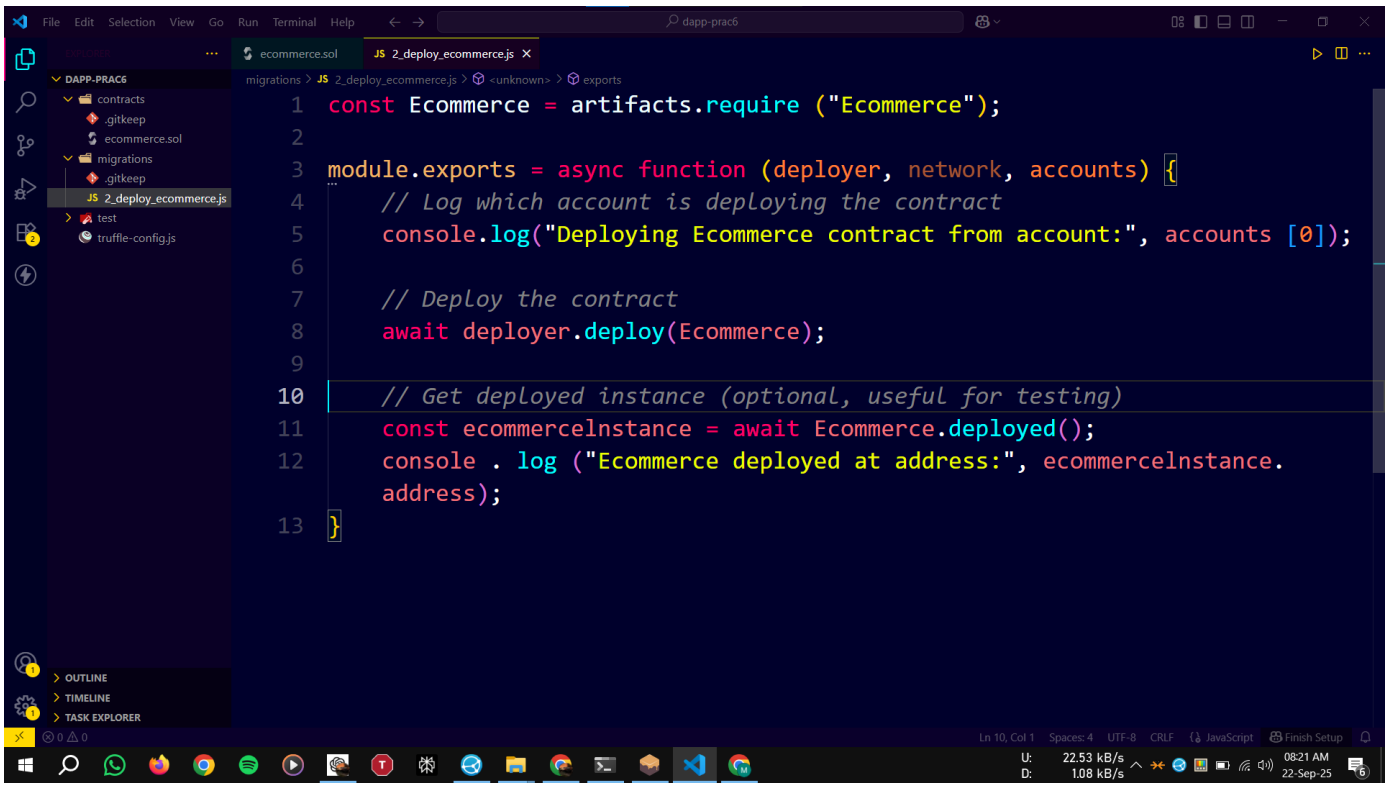
Now create a file in the contracts folder and name it as "*ecommerce.sol*" and write the smart contract into it.



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Ecommerce {
    address public contractOwner;
    uint public itemCount = 0;

    struct Item {
        uint id;
        string name;
        uint price; // in wei
        address payable owner;
        bool purchased;
    }

    mapping(uint => Item) public items;

    event ItemCreated(
        uint id,
        string name,
        uint price,
        address owner,
        bool purchased
    );
```

Step-3

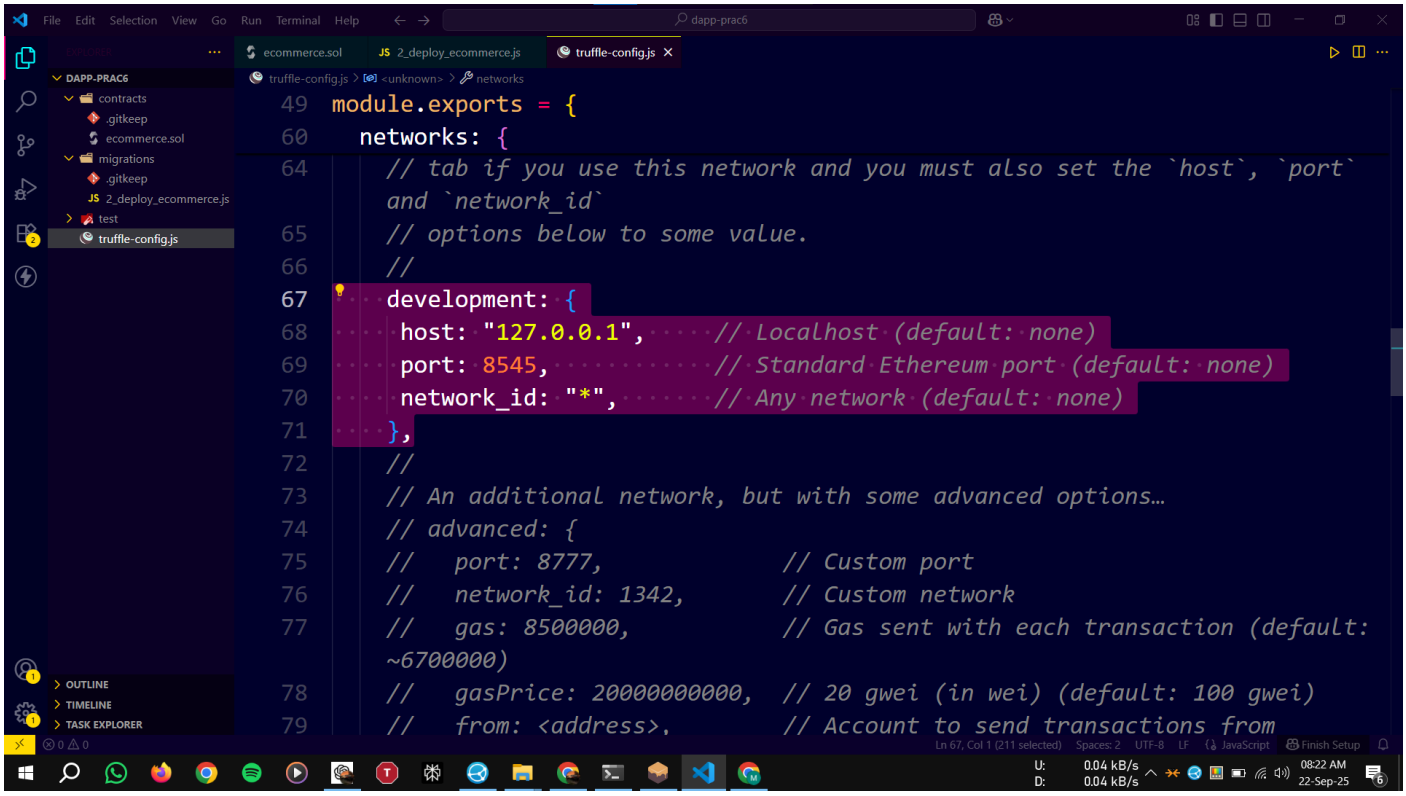
Now open migration folder and create a new file and name it "*2_deploy_ecommerce.js*" and write the following code into it.



```
const Ecommerce = artifacts.require ("Ecommerce");

module.exports = async function (deployer, network, accounts) {
    // Log which account is deploying the contract
    console.log("Deploying Ecommerce contract from account:", accounts [0]);

    // Deploy the contract
    await deployer.deploy(Ecommerce);

    // Get deployed instance (optional, useful for testing)
    const ecommerceInstance = await Ecommerce.deployed();
    console . log ("Ecommerce deployed at address:", ecommerceInstance.
    address);
}
```

Step-4

Now open *truffle-config.js* file and uncomment the following lines in that code as we did in the practical 5.



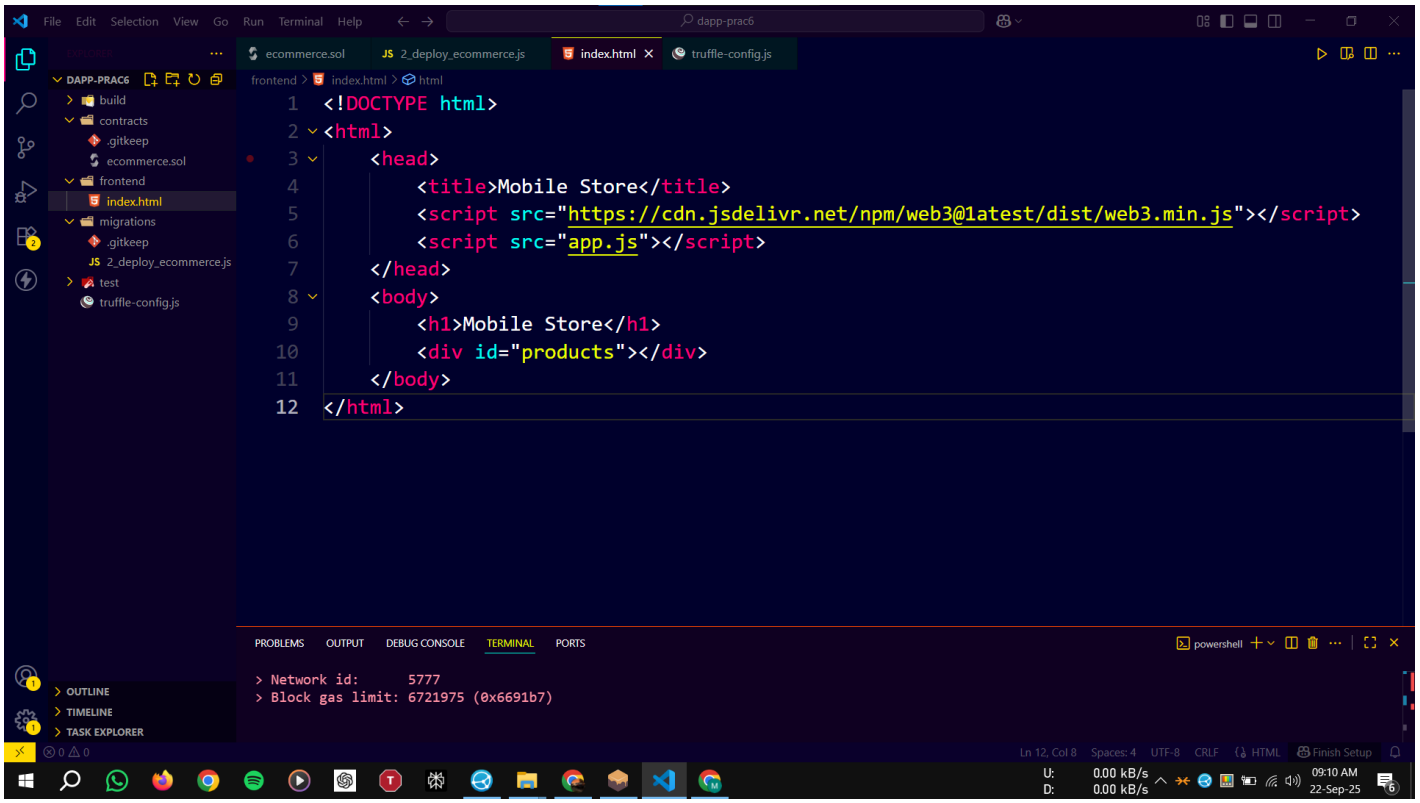
Step-5

Now open terminal and give the following command "*truffle compile*" and "*truffle migrate*" as follows.

```
====================
Deploying Ecommerce contract from account: 0x666fde2A68111715Cbf016be49752f2a7430E6Ad

   Replacing 'Ecommerce'
   ---------------------
   > transaction hash:    0x33816ccdbc1096a1d0f0aeb3b76fec0a2cc330ead4e369cf0782823db6dddd31
   > Blocks: 0            Seconds: 0
   > contract address:    0xf475da29974c79D8e2999Bb8eDa63bbB4033144b
   > block number:        2
   > block timestamp:     1757336989
   > account:             0x666fde2A68111715Cbf016be49752f2a7430E6Ad
   > balance:             99.993579745666715329
   > gas used:            962277 (0xeaee5)
   > gas price:           3.296939923 gwei
   > value sent:          0 ETH
   > total cost:          0.003172569458284671 ETH
```

Step-6

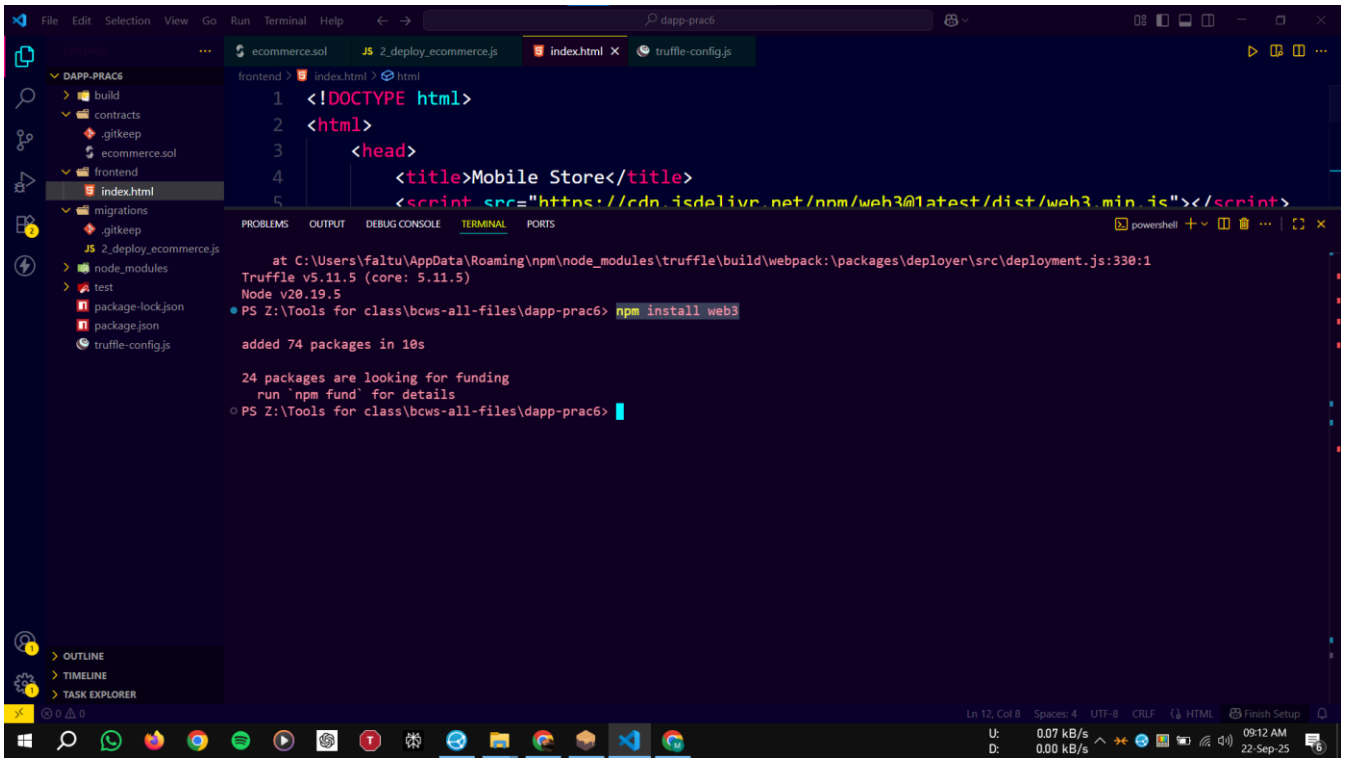
Now lets create the frontend for the Dapp • now lets create frontend floder an dnow create index.html file in I as follows.



Step-7

Now create styles.css file.

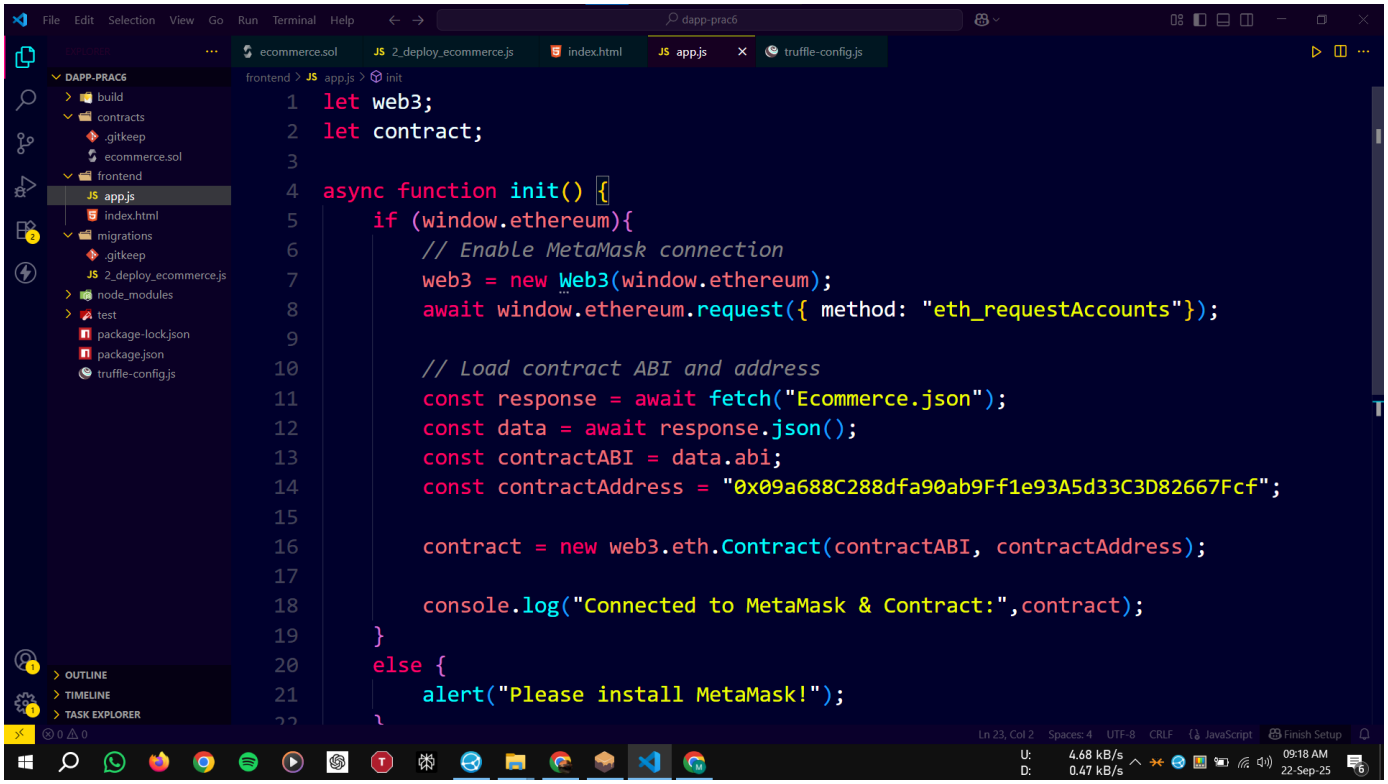
lets download web3 by using command *"npm install web3"* as follows



Step-9

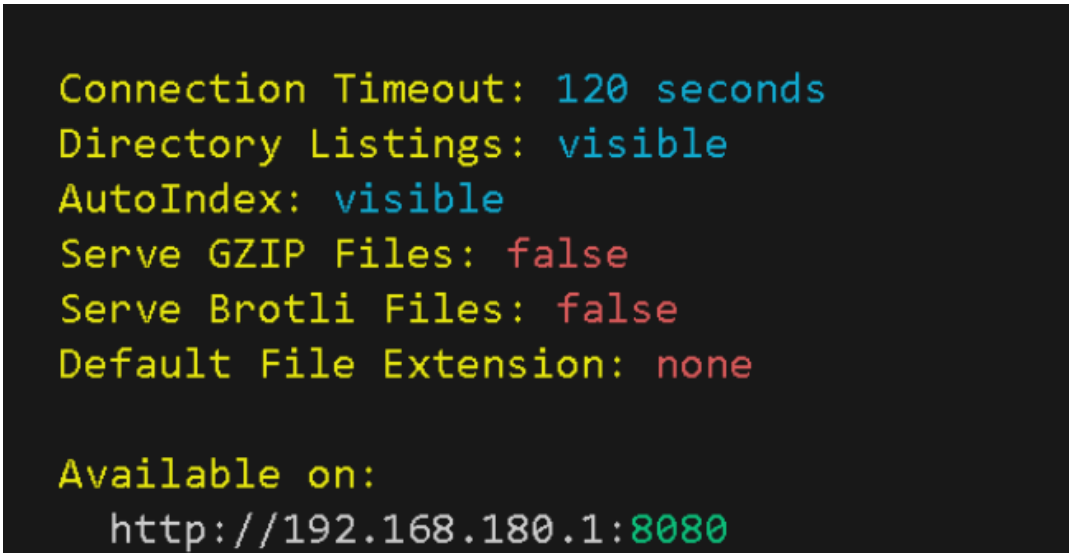
Now create app.js file.

In this file give the contract address which we had got in terminal we we migrated the truffle.



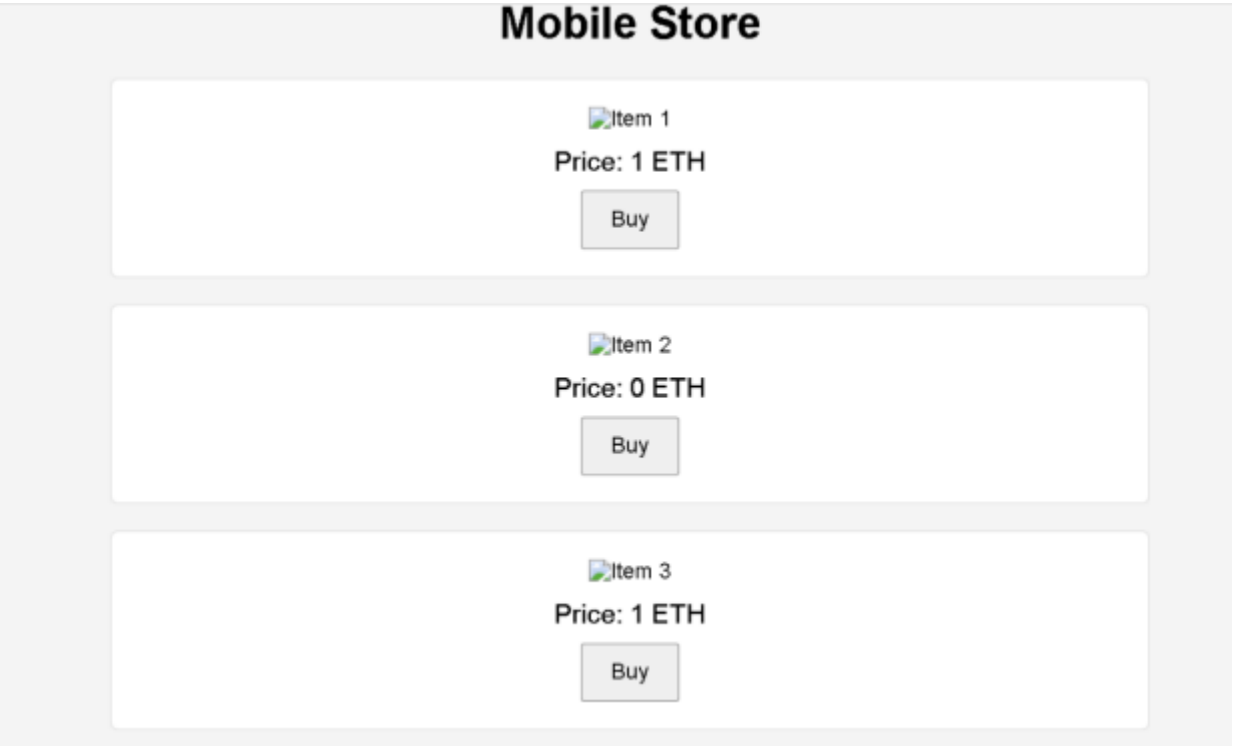
Step-10

Now open terminal and give command npx http-server to run the Dapp in browser



Step-11

Now click on the give links and open it now ouor webpage will be open



Conclusion:

In this practical we learn and perform how to Create a decentralized application (DApp) using a blockchain platform. And Deploy the DApp on a test network. and Interact with the DApp through a web interface.

Practical - 7

Aim: 1. Integrating blockchain functionality into web applications

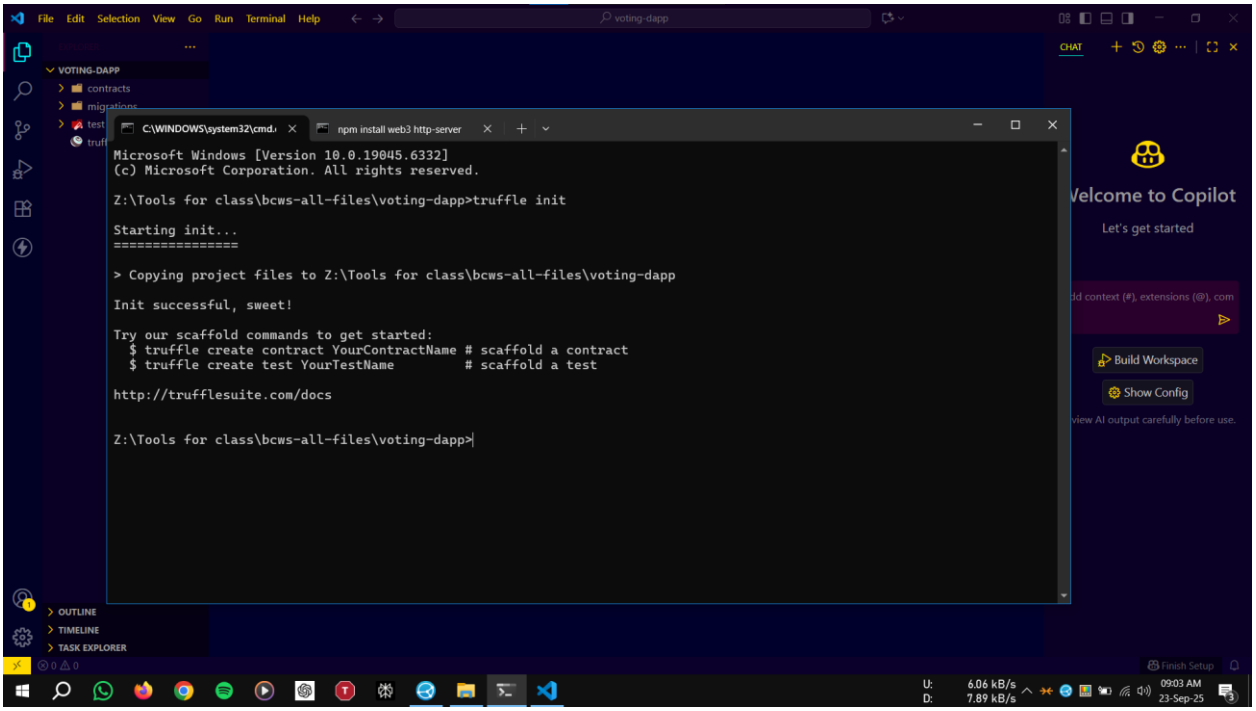
2. Implementing secure communication between web and blockchain components

Step1: Create a New Truffle Project

mkdir voting-dapp

cd voting-dapp

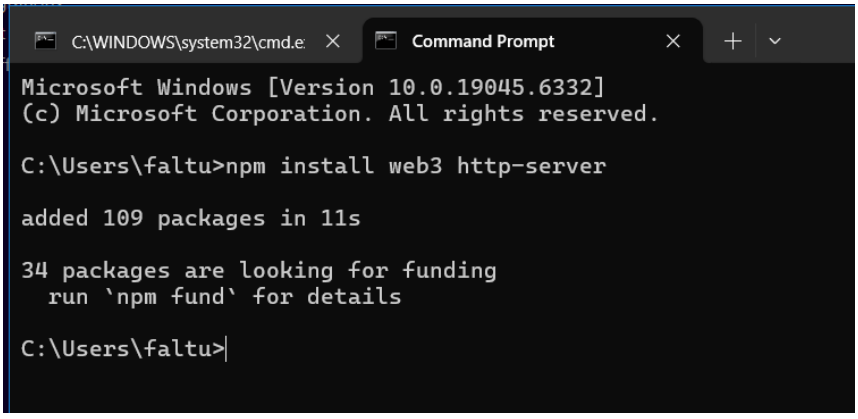
truffle init



Step2: Install Dependencies

Install `web3.js` and `http-server` for blockchain communication and serving the frontend

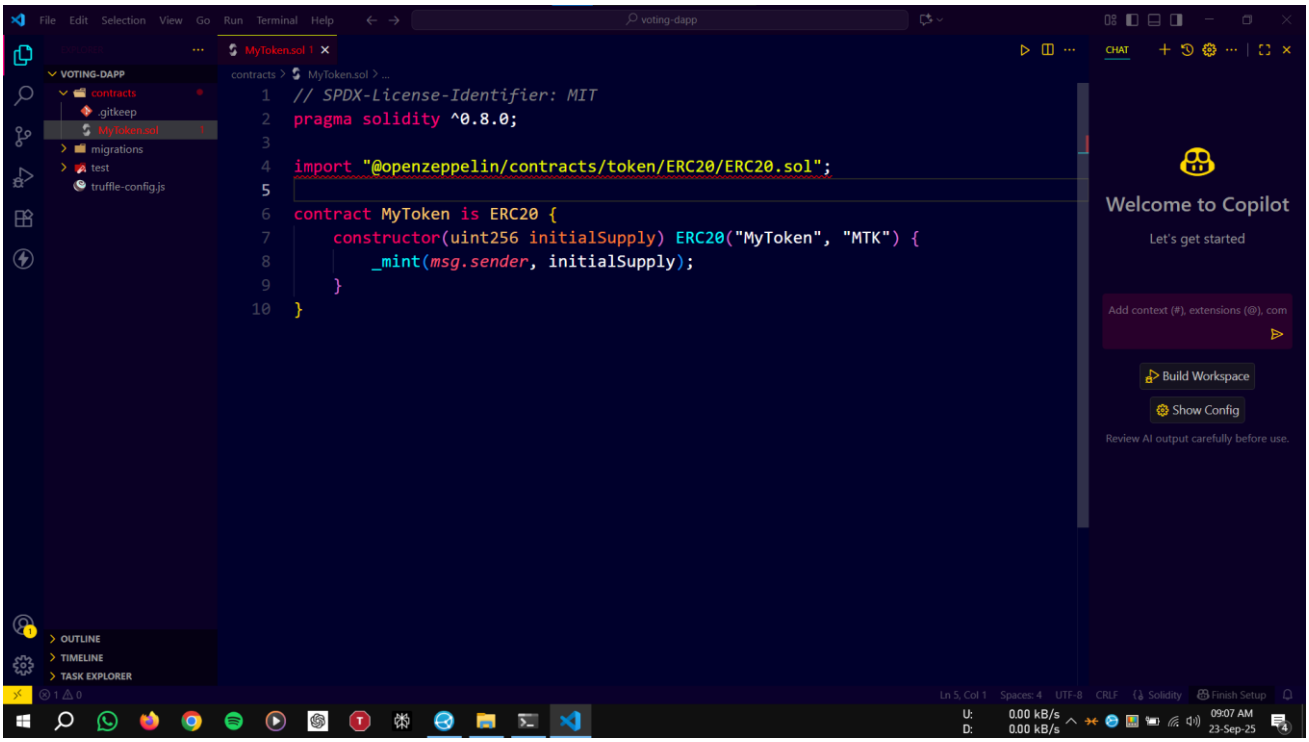
npm install web3 http-server



Step3: Create the Smart Contracts

Create a new file `*MyToken.sol*` in the `contracts` directory:

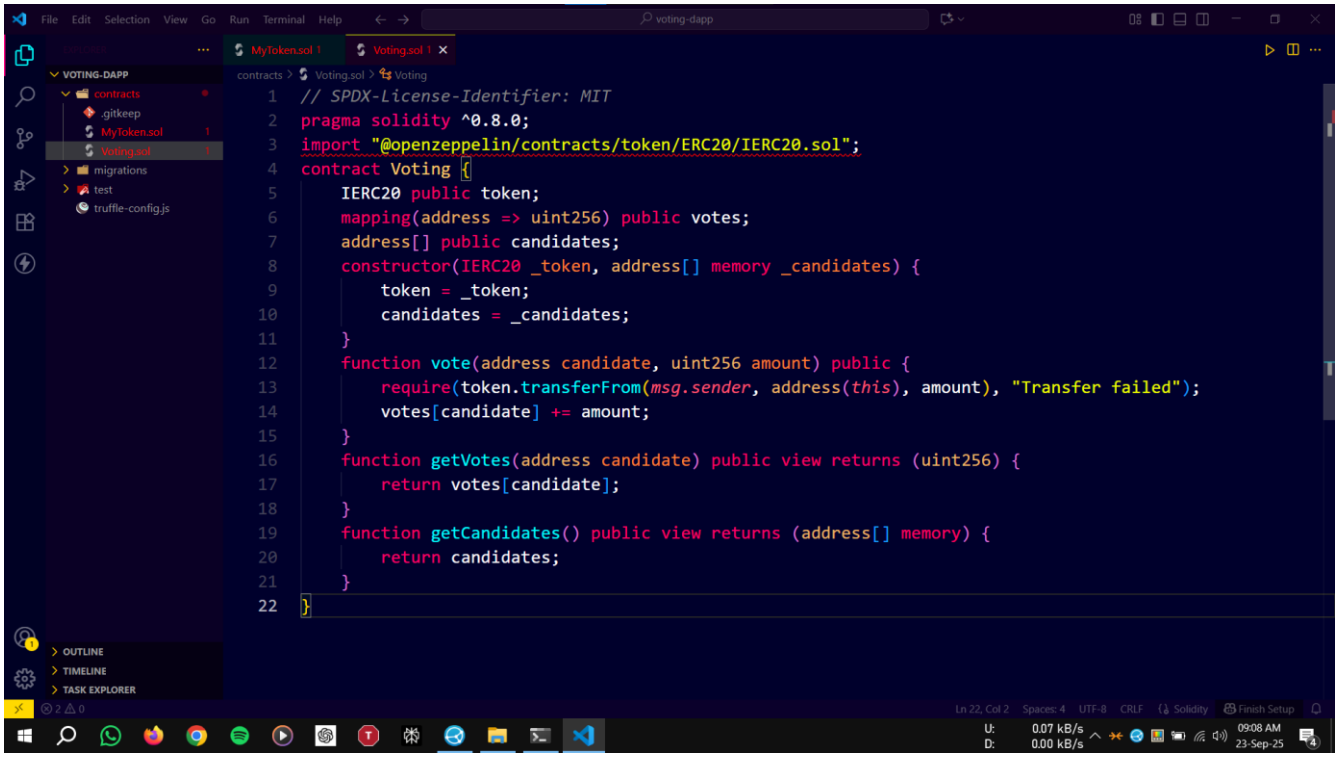
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {
        _mint(msg.sender, initialSupply);
    }
}
```



Create a new file `*Voting.sol*` in the `contracts` directory:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
contract Voting {
    IERC20 public token;
    mapping(address => uint256) public votes;
    address[] public candidates;
    constructor(IERC20 _token, address[] memory _candidates) {
        token = _token;
        candidates = _candidates;
```

```
    }
    function vote(address candidate, uint256 amount) public {
        require(token.transferFrom(msg.sender, address(this), amount), "Transfer failed");
        votes[candidate] += amount;
    }
    function getVotes(address candidate) public view returns (uint256) {
        return votes[candidate];
    }
    function getCandidates() public view returns (address[] memory) {
        return candidates;
    }
}
```

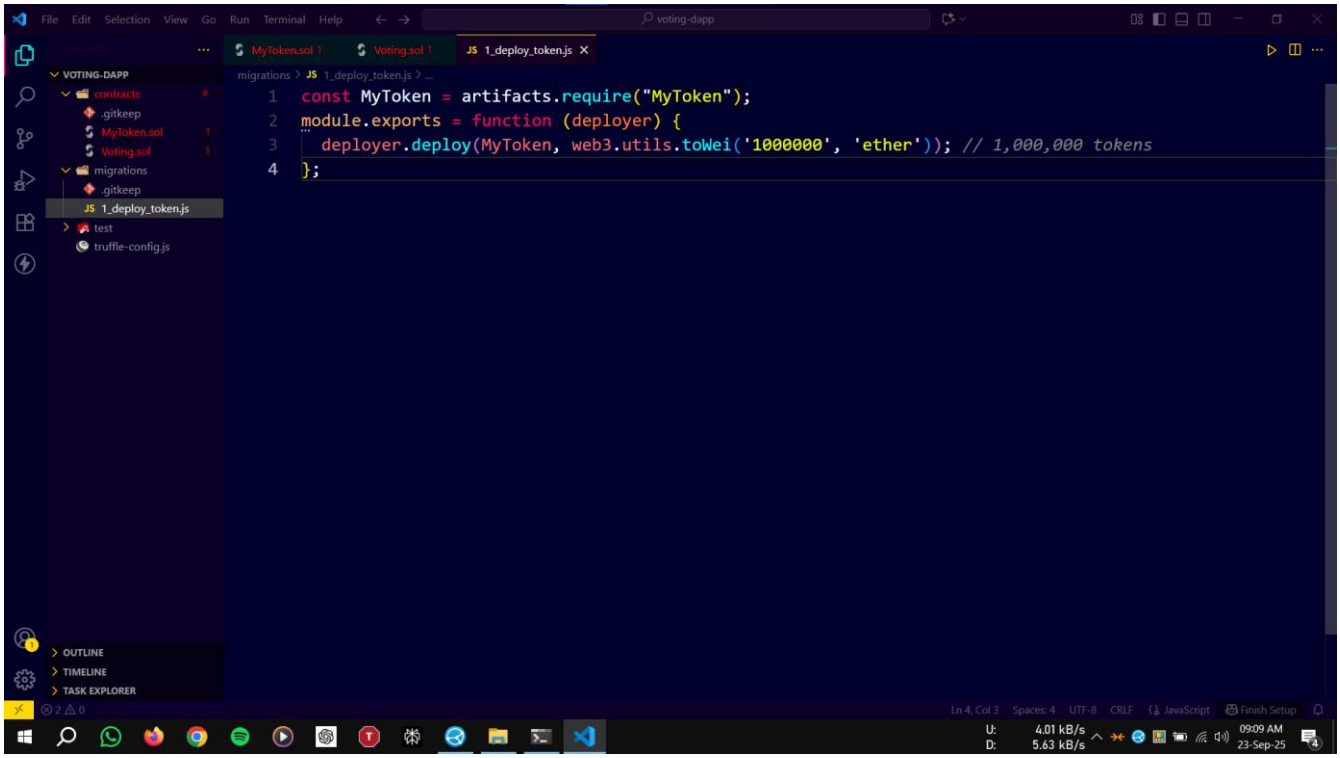


Step4: Write Migration Scripts.

Deploy Token (`1_deploy_token.js`)

Create a new file `*1_deploy_token.js*` in the `migrations` directory:

```
const MyToken = artifacts.require("MyToken");
module.exports = function (deployer) {
  deployer.deploy(MyToken, web3.utils.toWei('1000000', 'ether')); // 1,000,000 tokens
};
```



Deploy Voting Contract (`2_deploy_voting.js`)

Create a new file `2_deploy_voting.js` in the `migrations` directory:

const MyToken = artifacts.require("MyToken");

const Voting = artifacts.require("Voting");

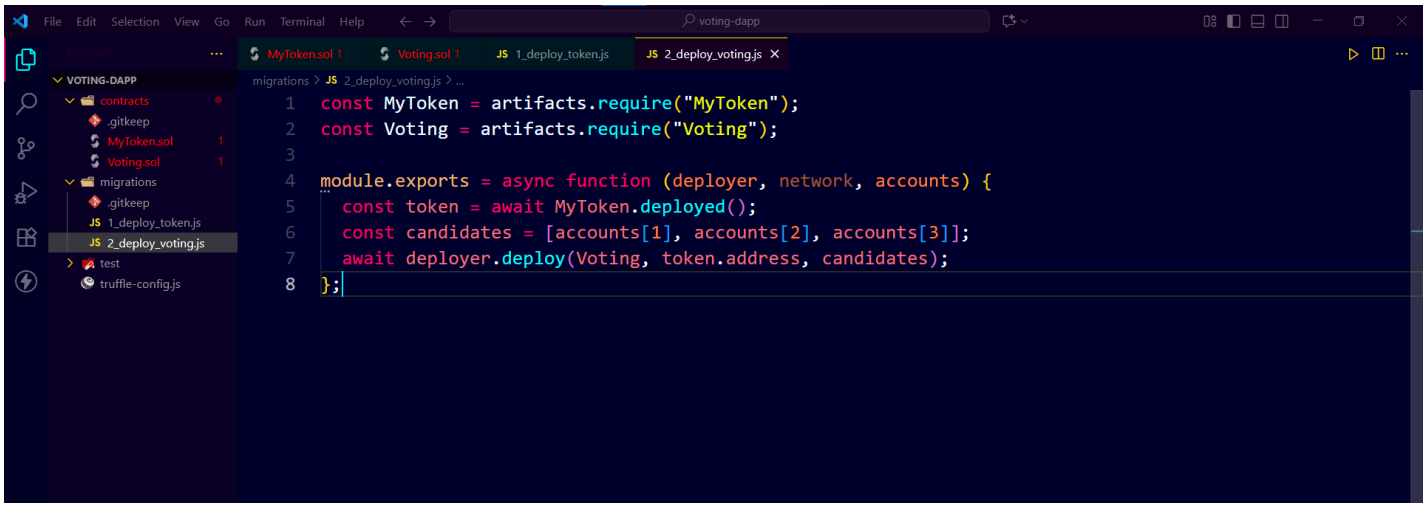
module.exports = async function (deployer, network, accounts) {

const token = await MyToken.deployed();

const candidates = [accounts[1], accounts[2], accounts[3]];

await deployer.deploy(Voting, token.address, candidates);

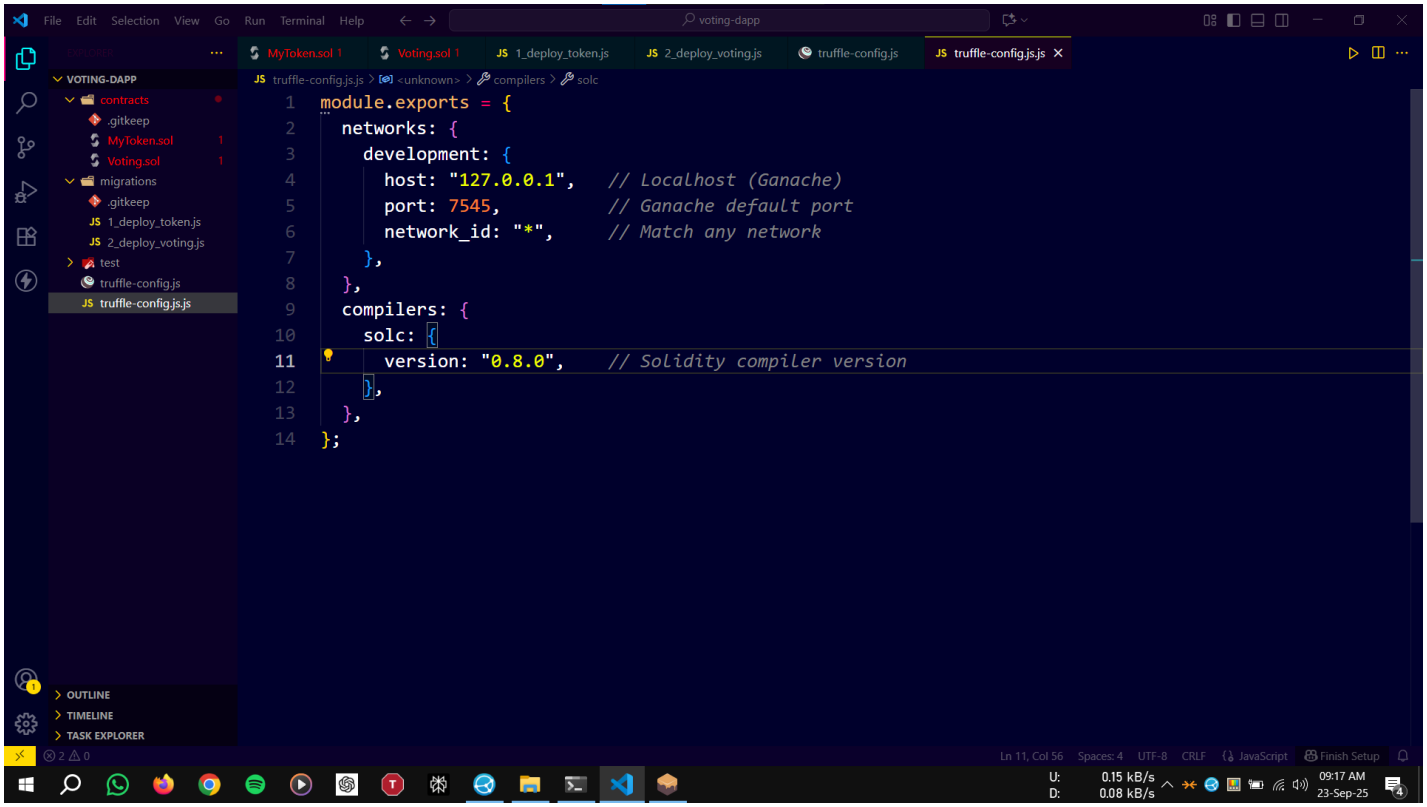
};



Step5: Configure Truffle.

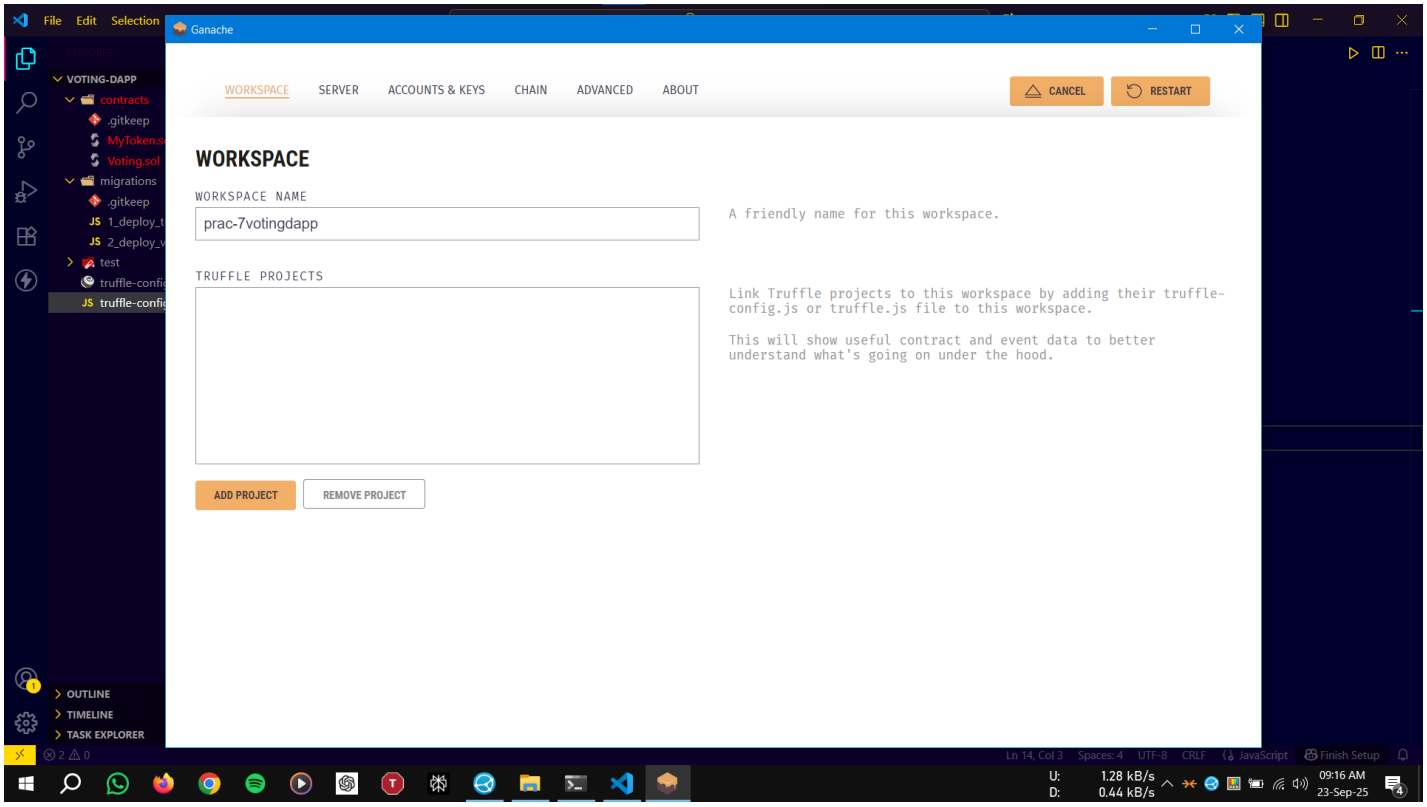
Ensure `truffle-config.js` is set up to use the local Ganache network:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",     // Localhost (Ganache)
      port: 7545,            // Ganache default port
      network_id: "*",       // Match any network
    },
  },
  compilers: {
    solc: {
      version: "0.8.0",      // Solidity compiler version
    },
  },
};
```



Step6: Start Ganache

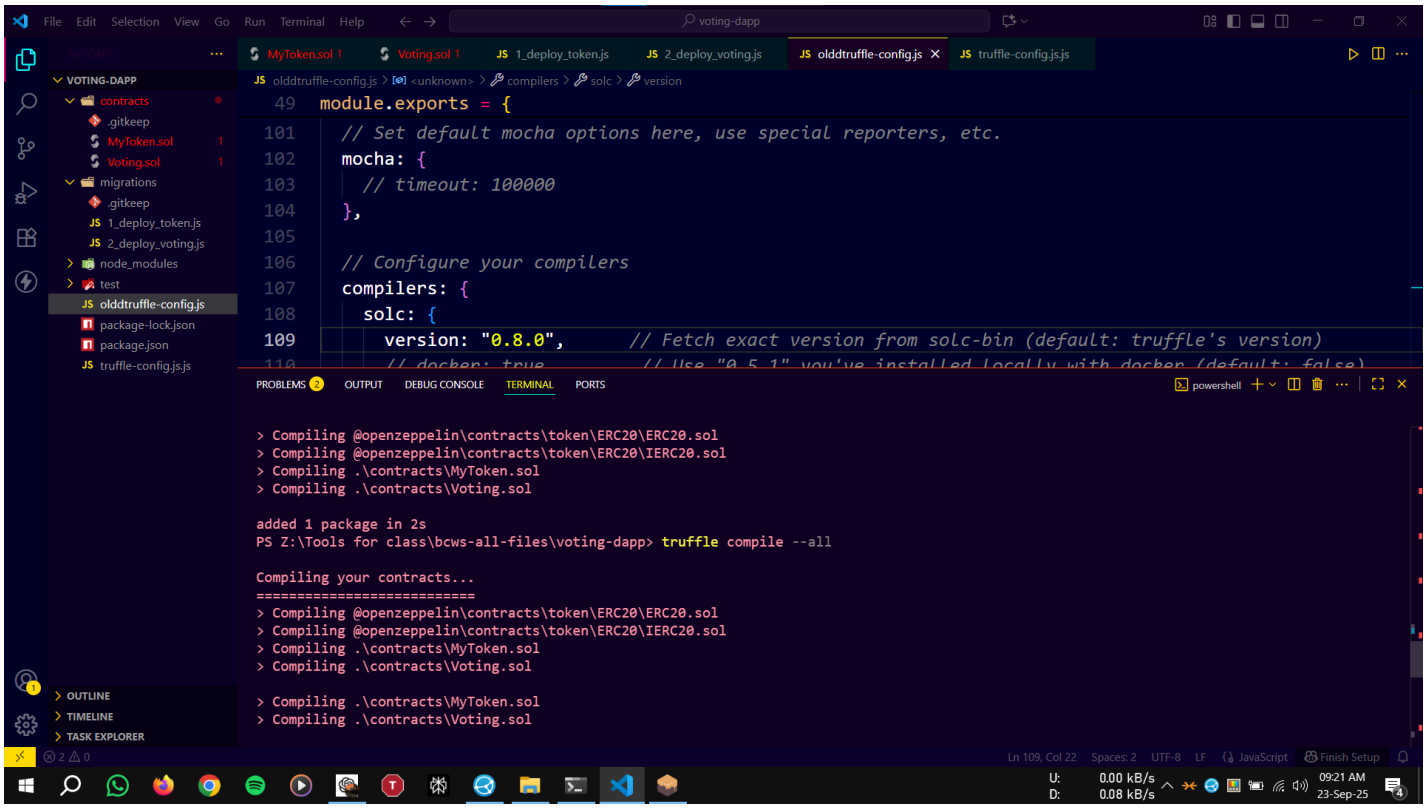
Open Ganache and start a new workspace. Ensure it's running on port.



Step7: Deploy the Contracts

Run the following command to deploy your contracts

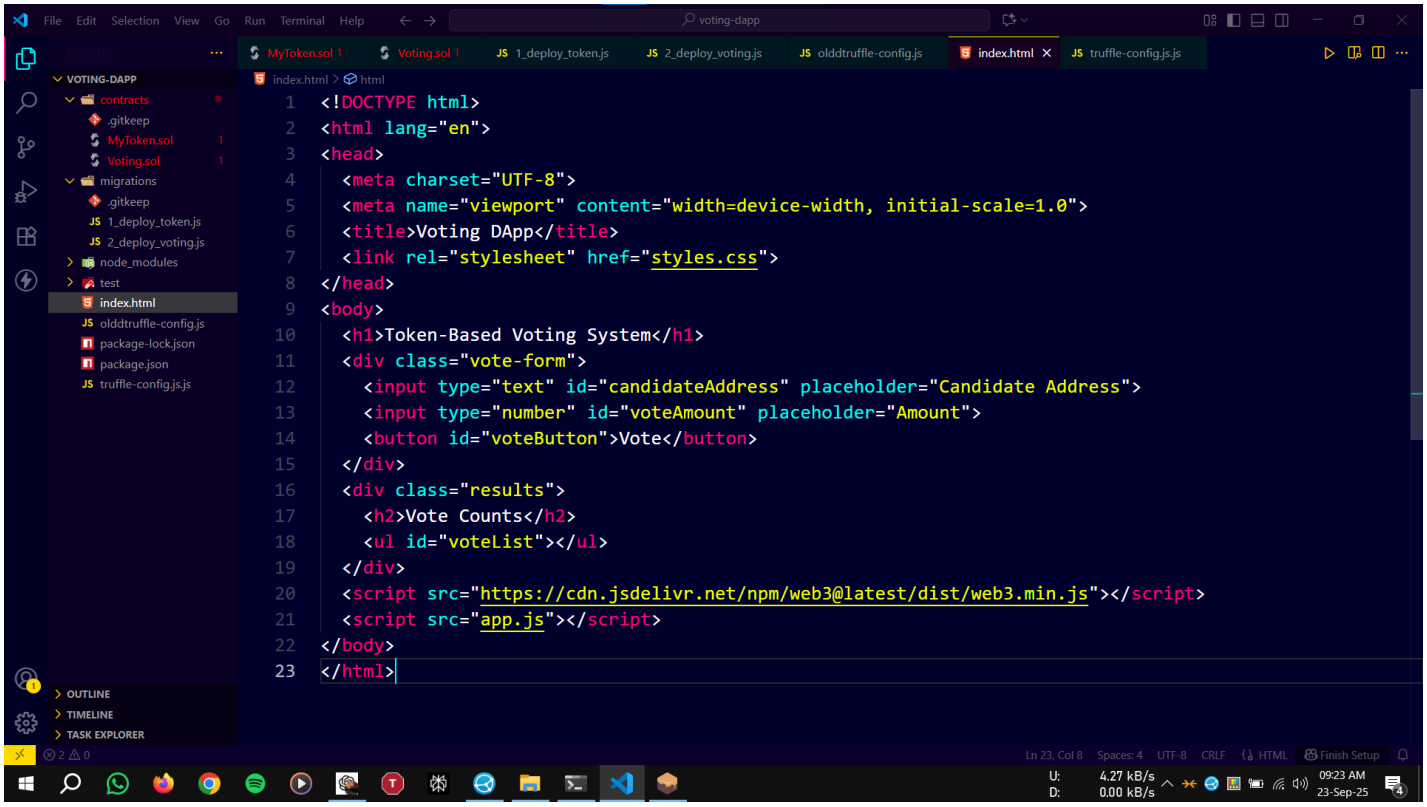
truffle migrate --network development



Step8: Create Frontend

Create index.html file:

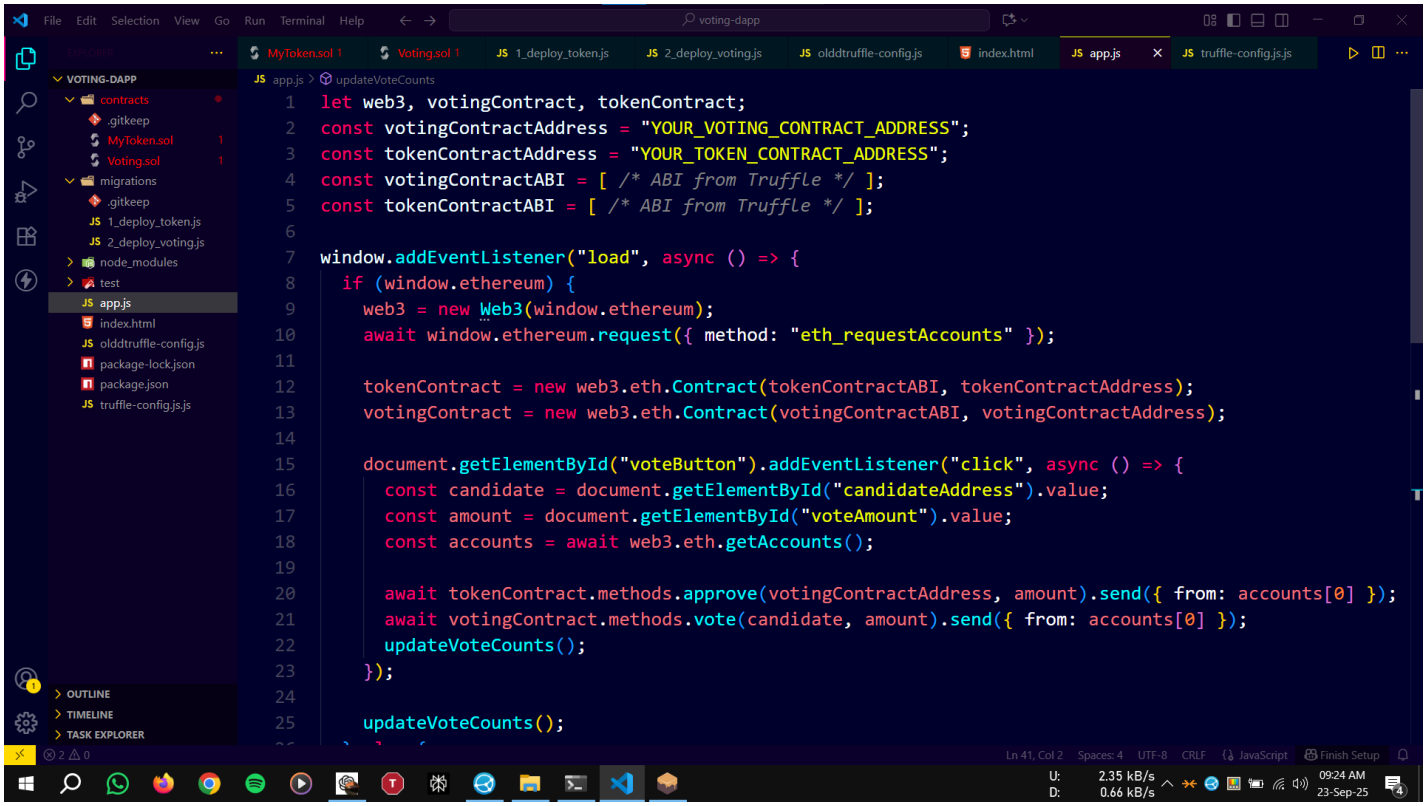
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Voting DApp</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Token-Based Voting System</h1>
    <div class="vote-form">
        <input type="text" id="candidateAddress" placeholder="Candidate Address">
        <input type="number" id="voteAmount" placeholder="Amount">
        <button id="voteButton">Vote</button>
    </div>
    <div class="results">
        <h2>Vote Counts</h2>
        <ul id="voteList"></ul>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/web3@latest/dist/web3.min.js"></script>
    <script src="app.js"></script>
</body>
</html>
```



Create app.js file:

```
let web3, votingContract, tokenContract;
const votingContractAddress = "YOUR_VOTING_CONTRACT_ADDRESS";
const tokenContractAddress = "YOUR_TOKEN_CONTRACT_ADDRESS";
const votingContractABI = [ /* ABI from Truffle */ ];
const tokenContractABI = [ /* ABI from Truffle */ ];

window.addEventListener("load", async () => {
    if (window.ethereum) {
        web3 = new Web3(window.ethereum);
        await window.ethereum.request({ method: "eth_requestAccounts" });

        tokenContract = new web3.eth.Contract(tokenContractABI, tokenContractAddress);
        votingContract = new web3.eth.Contract(votingContractABI, votingContractAddress);

        document.getElementById("voteButton").addEventListener("click", async () => {
            const candidate = document.getElementById("candidateAddress").value;
            const amount = document.getElementById("voteAmount").value;
            const accounts = await web3.eth.getAccounts();
```

```
            await tokenContract.methods.approve(votingContractAddress, amount).send({ from: accounts[0] });
            await votingContract.methods.vote(candidate, amount).send({ from: accounts[0] });
            updateVoteCounts();
        });

        updateVoteCounts();
    } else {
        alert("MetaMask not detected. Please install MetaMask.");
    }
});

async function updateVoteCounts() {
    const candidates = await votingContract.methods.getCandidates().call();
    const voteList = document.getElementById("voteList");
    voteList.innerHTML = "";
    for (let i = 0; i < candidates.length; i++) {
        const votes = await votingContract.methods.getVotes(candidates[i]).call();
        const listItem = document.createElement("li");
        listItem.textContent = `${candidates[i]}: ${votes} tokens`;
        voteList.appendChild(listItem);
    }
}
```

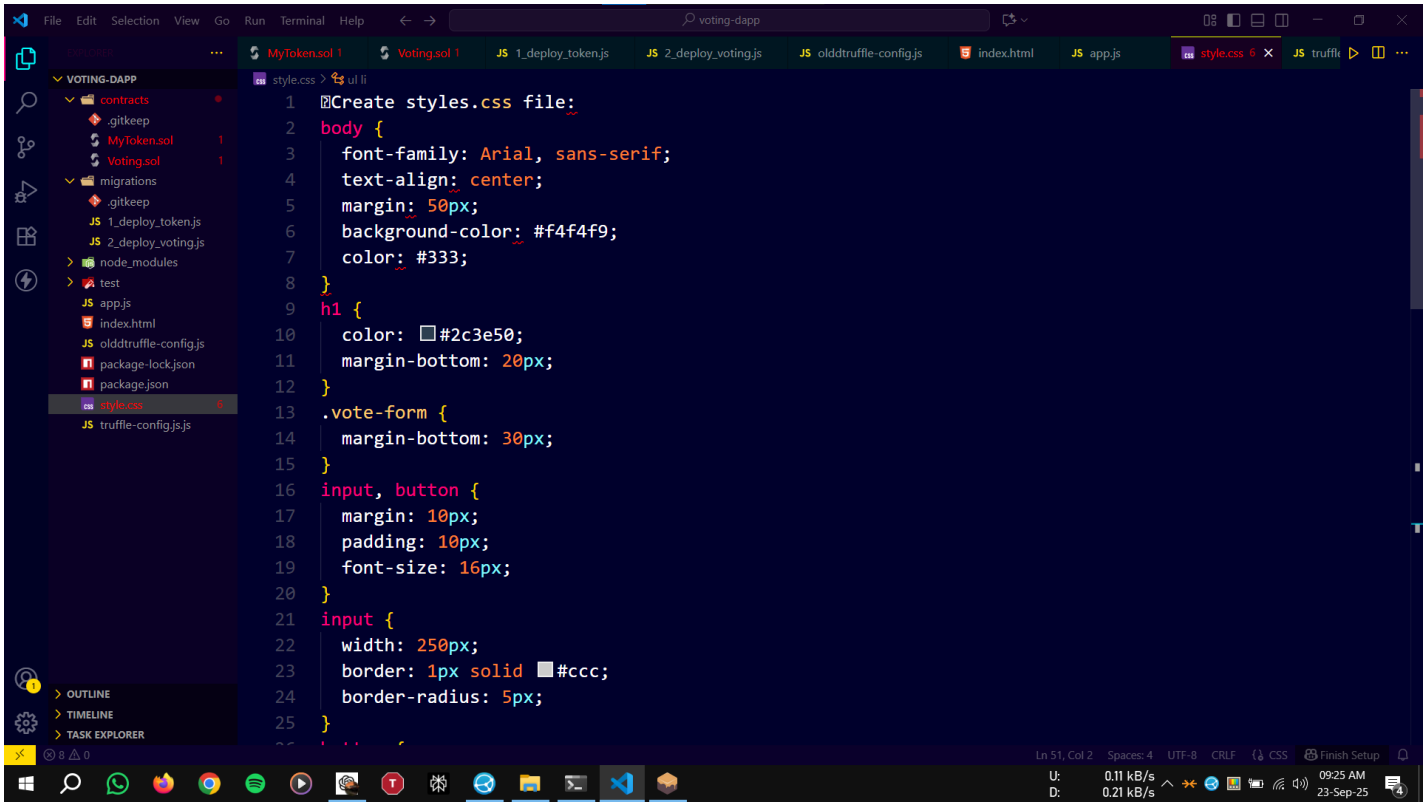


```
let web3, votingContract, tokenContract;
const votingContractAddress = "YOUR_VOTING_CONTRACT_ADDRESS";
const tokenContractAddress = "YOUR_TOKEN_CONTRACT_ADDRESS";
const votingContractABI = [ /* ABI from Truffle */ ];
const tokenContractABI = [ /* ABI from Truffle */ ];

window.addEventListener("load", async () => {
  if (window.ethereum) {
    web3 = new Web3(window.ethereum);
    await window.ethereum.request({ method: "eth_requestAccounts" });

    tokenContract = new web3.eth.Contract(tokenContractABI, tokenContractAddress);
    votingContract = new web3.eth.Contract(votingContractABI, votingContractAddress);

    document.getElementById("voteButton").addEventListener("click", async () => {
      const candidate = document.getElementById("candidateAddress").value;
      const amount = document.getElementById("voteAmount").value;
      const accounts = await web3.eth.getAccounts();

      await tokenContract.methods.approve(votingContractAddress, amount).send({ from: accounts[0] });
      await votingContract.methods.vote(candidate, amount).send({ from: accounts[0] });
      updateVoteCounts();
    });

    updateVoteCounts();
```

Create styles.css file:

```
body {
    font-family: Arial, sans-serif;
    text-align: center;
    margin: 50px;
    background-color: #f4f4f9;
    color: #333;
}
h1 {
    color: #2c3e50;
    margin-bottom: 20px;
}
.vote-form {
    margin-bottom: 30px;
}
input, button {
    margin: 10px;
    padding: 10px;
    font-size: 16px;
}
```

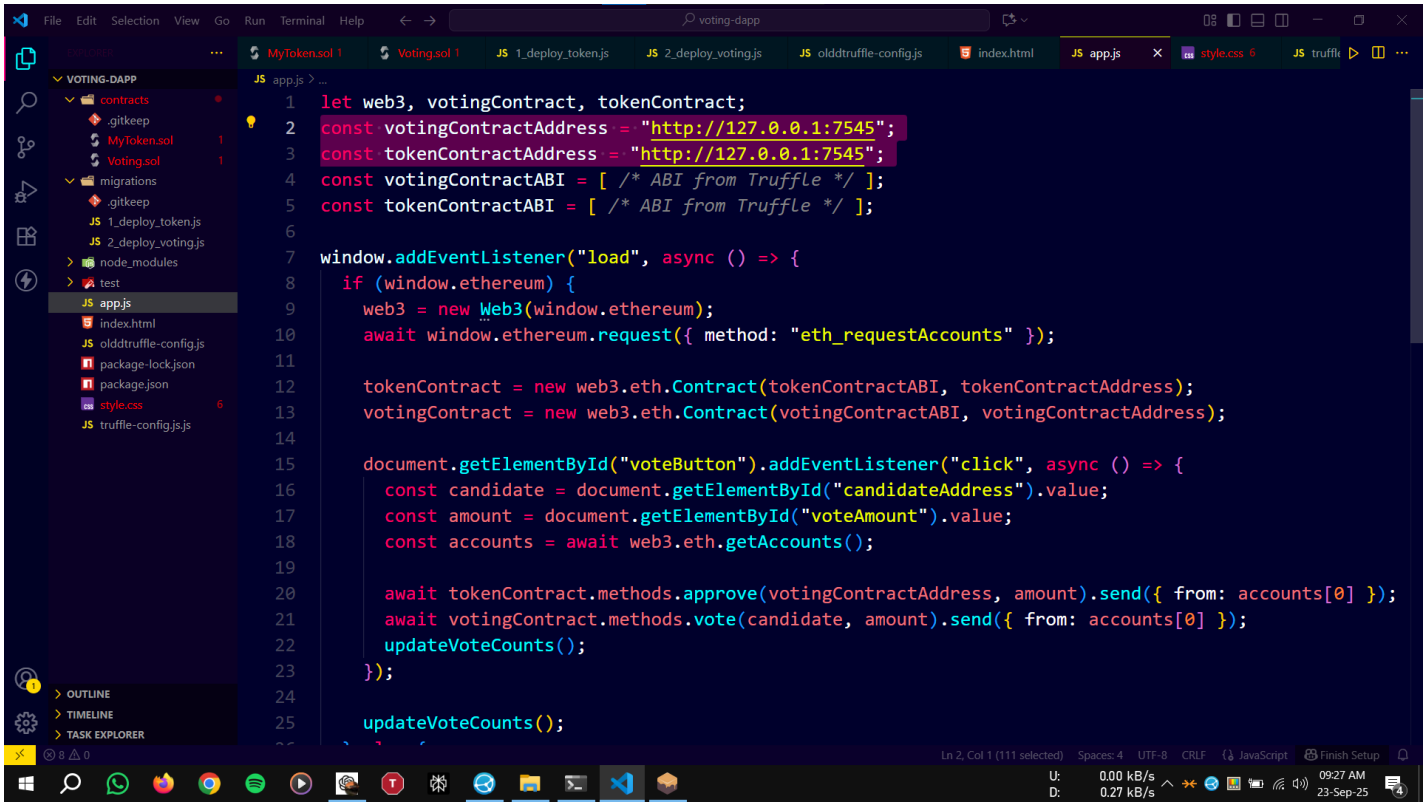
```
input {
    width: 250px;
    border: 1px solid #ccc;
    border-radius: 5px;
}
button {
    background-color: #3498db;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    transition: background 0.3s ease;
}
button:hover {
    background-color: #2980b9;
}
.results {
    margin-top: 30px;
}
ul {
    list-style-type: none;
    padding: 0;
}
ul li {
    background: #ecf0f1;
    margin: 8px auto;
    padding: 12px;
    border-radius: 5px;
    width: 300px;
    text-align: center;
}
```



```
Create styles.css file:
body {
  font-family: Arial, sans-serif;
  text-align: center;
  margin: 50px;
  background-color: #f4f4f9;
  color: #333;
}
h1 {
  color: #2c3e50;
  margin-bottom: 20px;
}
.vote-form {
  margin-bottom: 30px;
}
input, button {
  margin: 10px;
  padding: 10px;
  font-size: 16px;
}
input {
  width: 250px;
  border: 1px solid #ccc;
  border-radius: 5px;
}
```

Step9: Get Contract ABIs

After deploying the contracts, retrieve the ABI from `build/contracts` for both `voting` and `mytoken` contracts. Replace the placeholders in `app.js` with the actual ABIs and contract addresses.

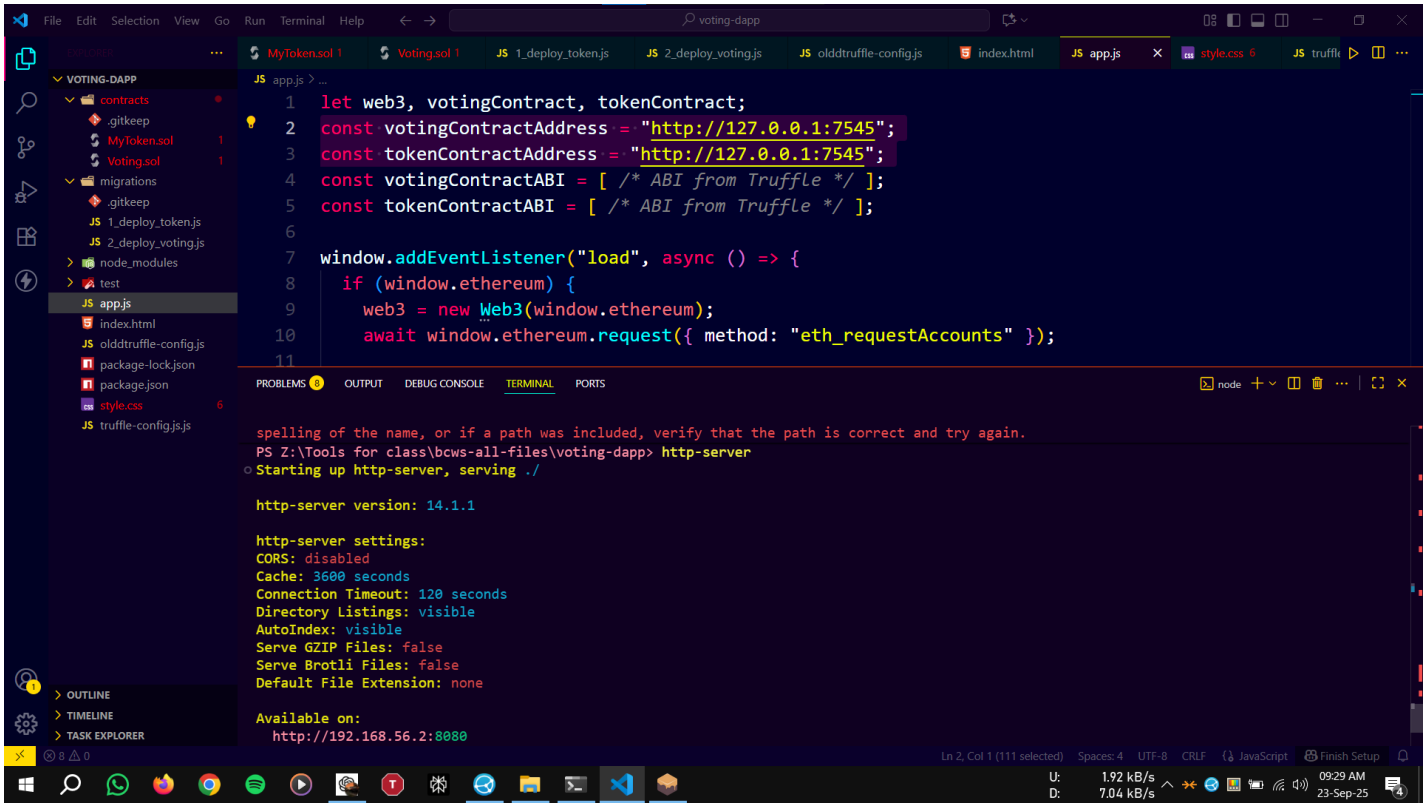


```
let web3, votingContract, tokenContract;
const votingContractAddress = "http://127.0.0.1:7545";
const tokenContractAddress = "http://127.0.0.1:7545";
const votingContractABI = [ /* ABI from Truffle */ ];
const tokenContractABI = [ /* ABI from Truffle */ ];

window.addEventListener("load", async () => {
  if (window.ethereum) {
    web3 = new Web3(window.ethereum);
    await window.ethereum.request({ method: "eth_requestAccounts" });

    tokenContract = new web3.eth.Contract(tokenContractABI, tokenContractAddress);
    votingContract = new web3.eth.Contract(votingContractABI, votingContractAddress);

    document.getElementById("voteButton").addEventListener("click", async () => {
      const candidate = document.getElementById("candidateAddress").value;
      const amount = document.getElementById("voteAmount").value;
      const accounts = await web3.eth.getAccounts();

      await tokenContract.methods.approve(votingContractAddress, amount).send({ from: accounts[0] });
      await votingContract.methods.vote(candidate, amount).send({ from: accounts[0] });
      updateVoteCounts();
    });

    updateVoteCounts();
```

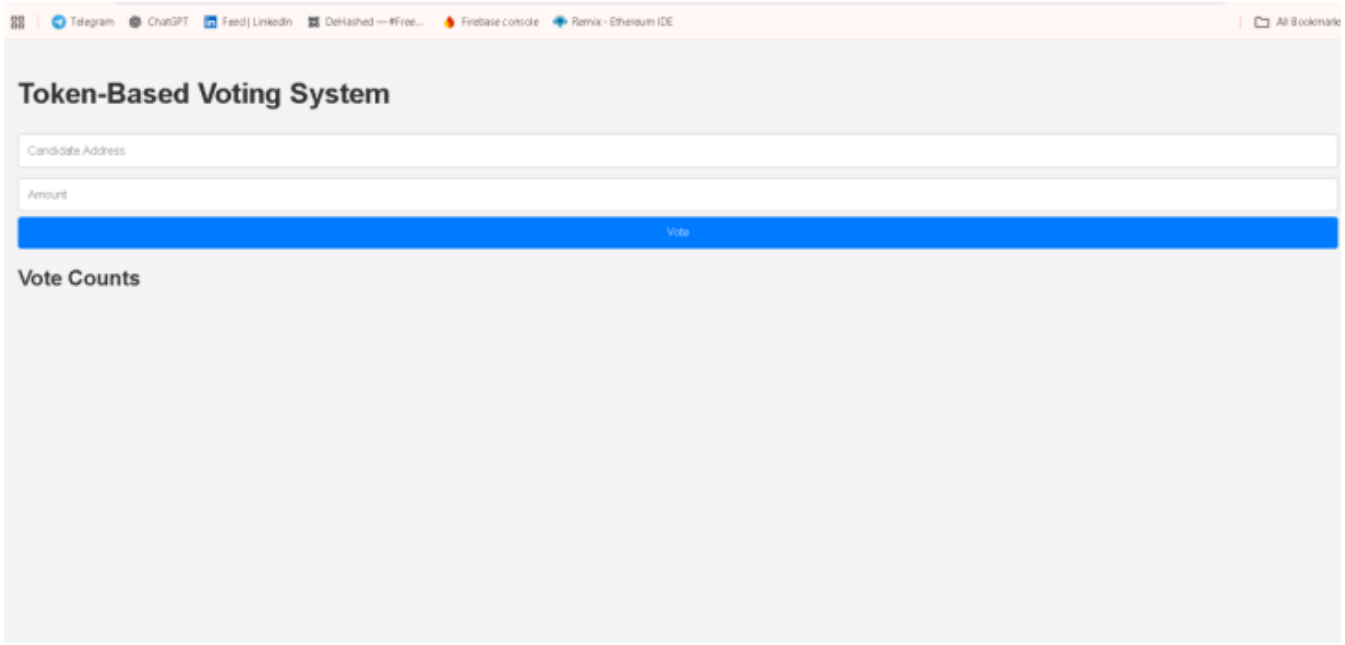
Step10: Run the Frontend

Use `http-server` to serve your frontend

http-server



Step11: Open the browser and navigate to ` http://127.0.0.1:8081`. Connect MetaMask and interact with the voting dApp.



Conclusion:

We have successfully deployed Voting DApp.

Practical - 8

Aim: Securing Decentralized Identity

1. Exploring decentralized identity solutions

2. Implementing secure identity management on the blockchain

Step1: Setup

Install Node.js, npm, Truffle, Ganache, MetaMask, and http-server.

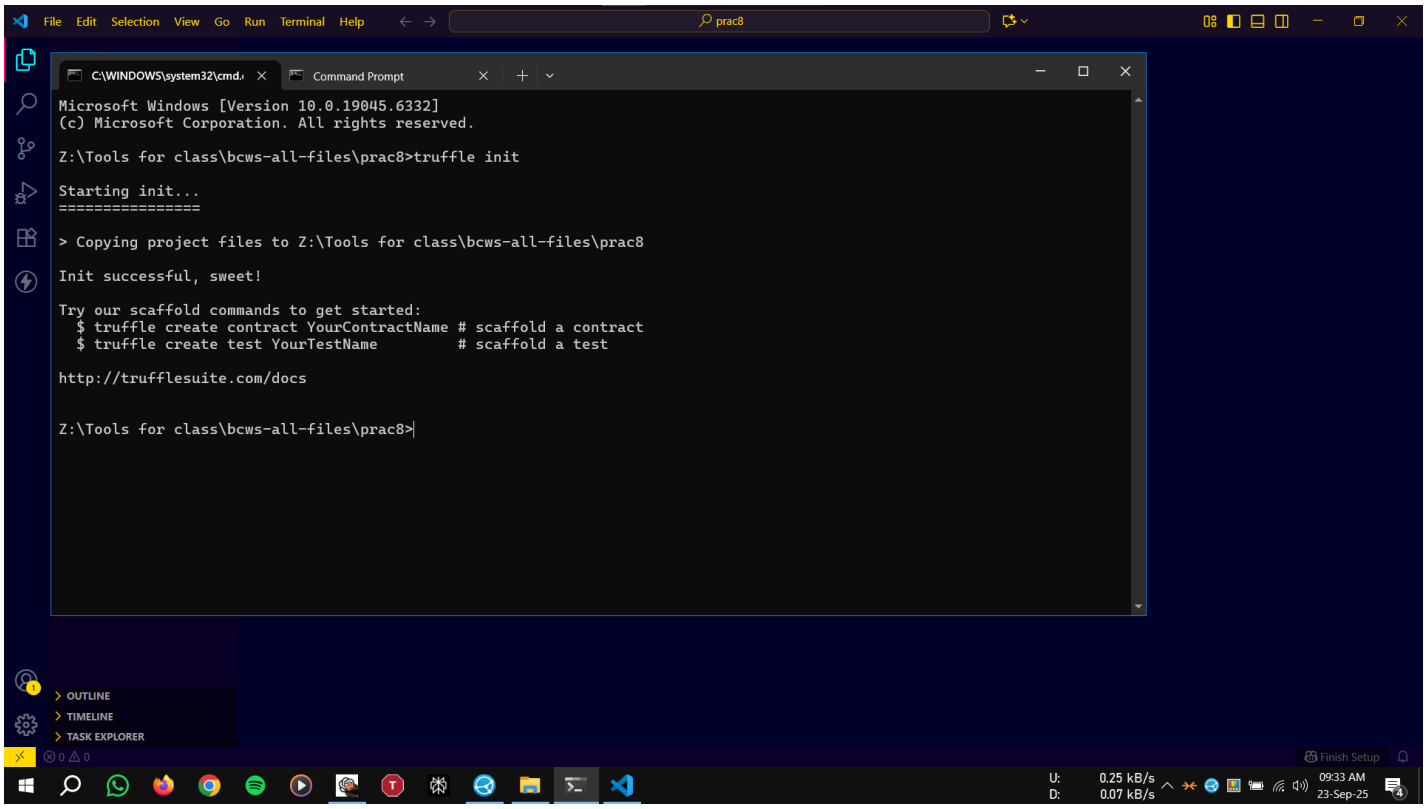
Create project:

mkdir prac8

cd prac8

Initialize truffle :

truffle init



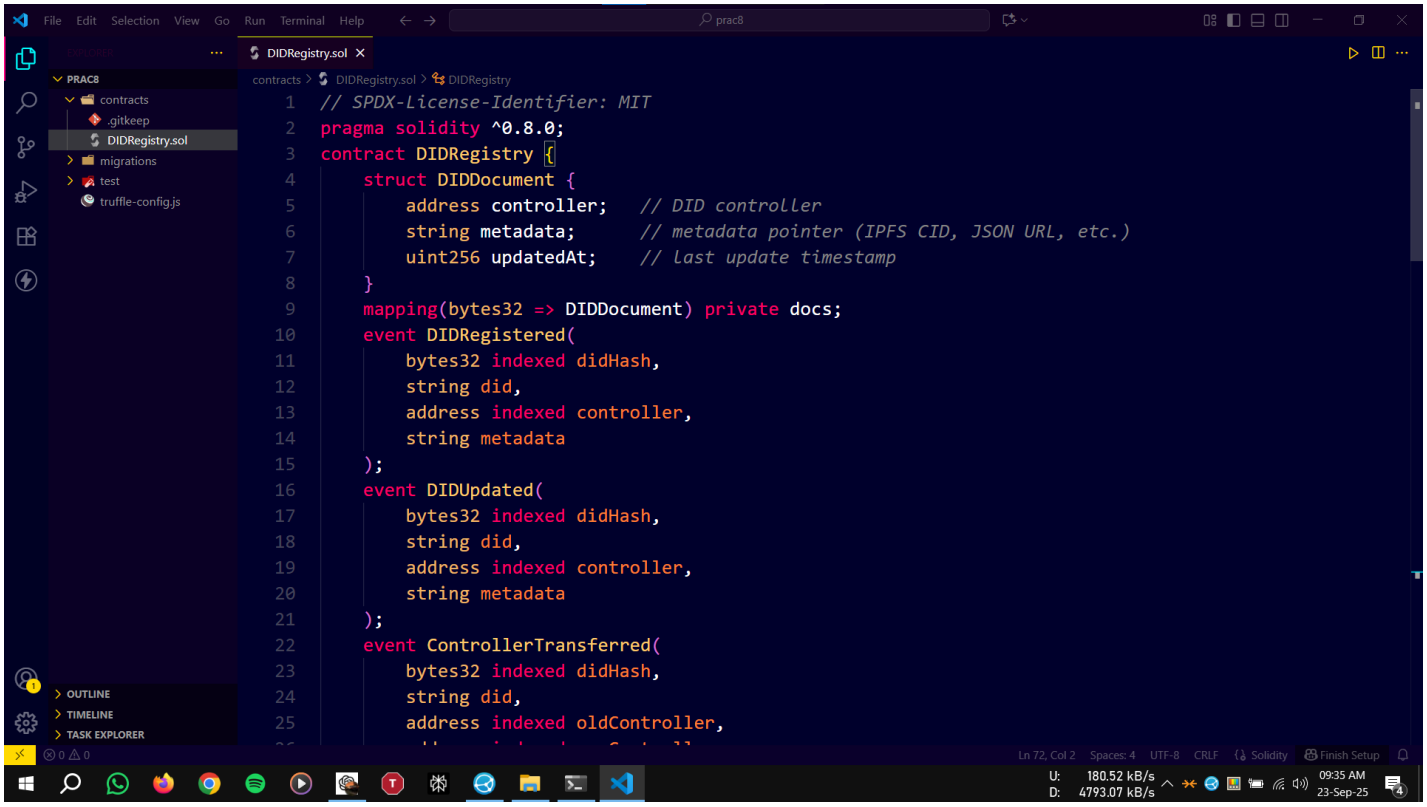
Step2: Smart Contract

Create DIDRegistry.sol in contracts directory.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract DIDRegistry {
    struct DIDDocument {
        address controller;      // DID controller
```

```
        string metadata;        // metadata pointer (IPFS CID, JSON URL, etc.)
        uint256 updatedAt;      // last update timestamp
    }
    mapping(bytes32 => DIDDocument) private docs;
    event DIDRegistered(
        bytes32 indexed didHash,
        string did,
        address indexed controller,
        string metadata
    );
    event DIDUpdated(
        bytes32 indexed didHash,
        string did,
        address indexed controller,
        string metadata
    );
    event ControllerTransferred(
        bytes32 indexed didHash,
        string did,
        address indexed oldController,
        address indexed newController
    );
    modifier onlyController(bytes32 h) {
        require(docs[h].controller == msg.sender, "Caller is not controller");
        _;
    }
    function registerDID(string calldata did, string calldata metadata) external {
        bytes32 h = keccak256(abi.encodePacked(did));
        require(docs[h].controller == address(0), "DID already registered");
        docs[h] = DIDDocument({
            controller: msg.sender,
            metadata: metadata,
            updatedAt: block.timestamp
        });
        emit DIDRegistered(h, did, msg.sender, metadata);
```

```
    }
    function updateMetadata(string calldata did, string calldata metadata) external {
        bytes32 h = keccak256(abi.encodePacked(did));
        require(docs[h].controller != address(0), "DID not registered");
        require(docs[h].controller == msg.sender, "Caller is not controller");
        docs[h].metadata = metadata;
        docs[h].updatedAt = block.timestamp;
        emit DIDUpdated(h, did, msg.sender, metadata);
    }
    function transferController(string calldata did, address newController) external {
        bytes32 h = keccak256(abi.encodePacked(did));
        require(docs[h].controller != address(0), "DID not registered");
        require(docs[h].controller == msg.sender, "Caller is not controller");
        address oldController = docs[h].controller;
        docs[h].controller = newController;
        docs[h].updatedAt = block.timestamp;
        emit ControllerTransferred(h, did, oldController, newController);
    }
    function getDID(string calldata did) external view returns (
        address controller,
        string memory metadata,
        uint256 updatedAt
    ) {
        bytes32 h = keccak256(abi.encodePacked(did));
        DIDDocument memory d = docs[h];
        return (d.controller, d.metadata, d.updatedAt);
    }
    function controllerOf(string calldata did) external view returns (address) {
        bytes32 h = keccak256(abi.encodePacked(did));
        return docs[h].controller;
    }
}
```



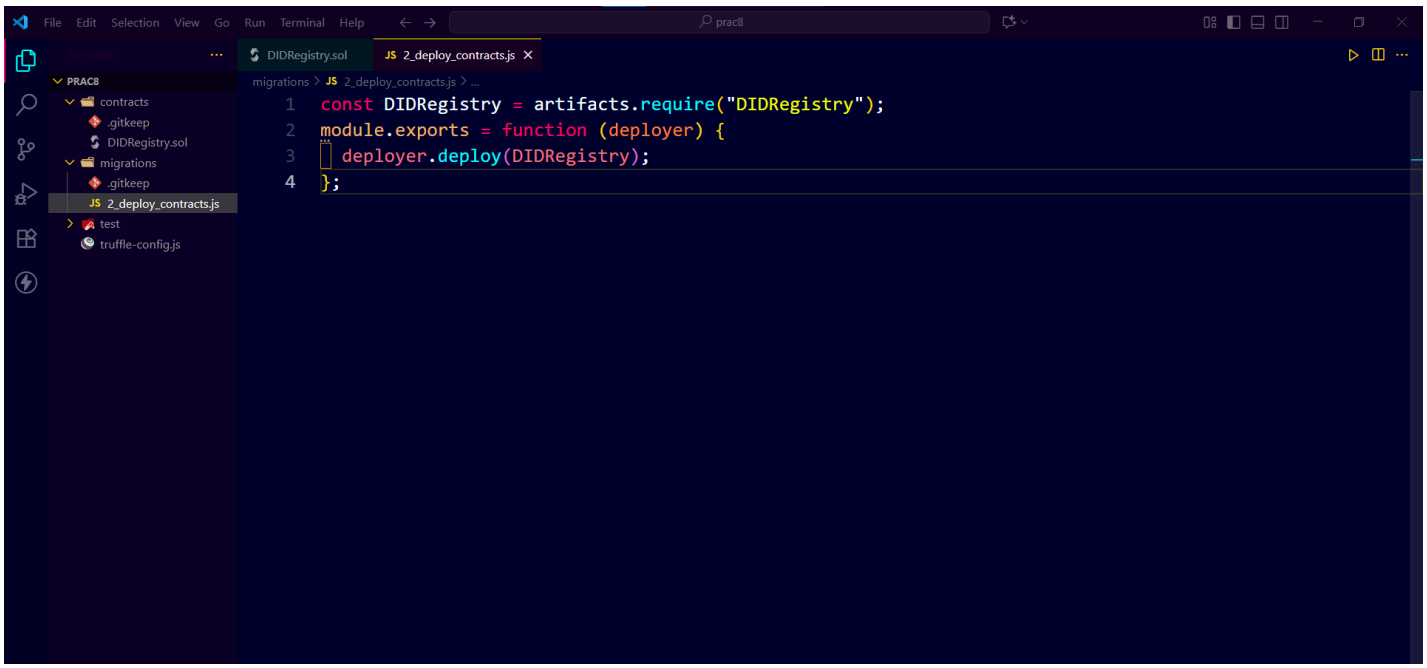
Create 2_deploy_contracts.js in migrations folder:

const DIDRegistry = artifacts.require("DIDRegistry");

module.exports = function (deployer) {

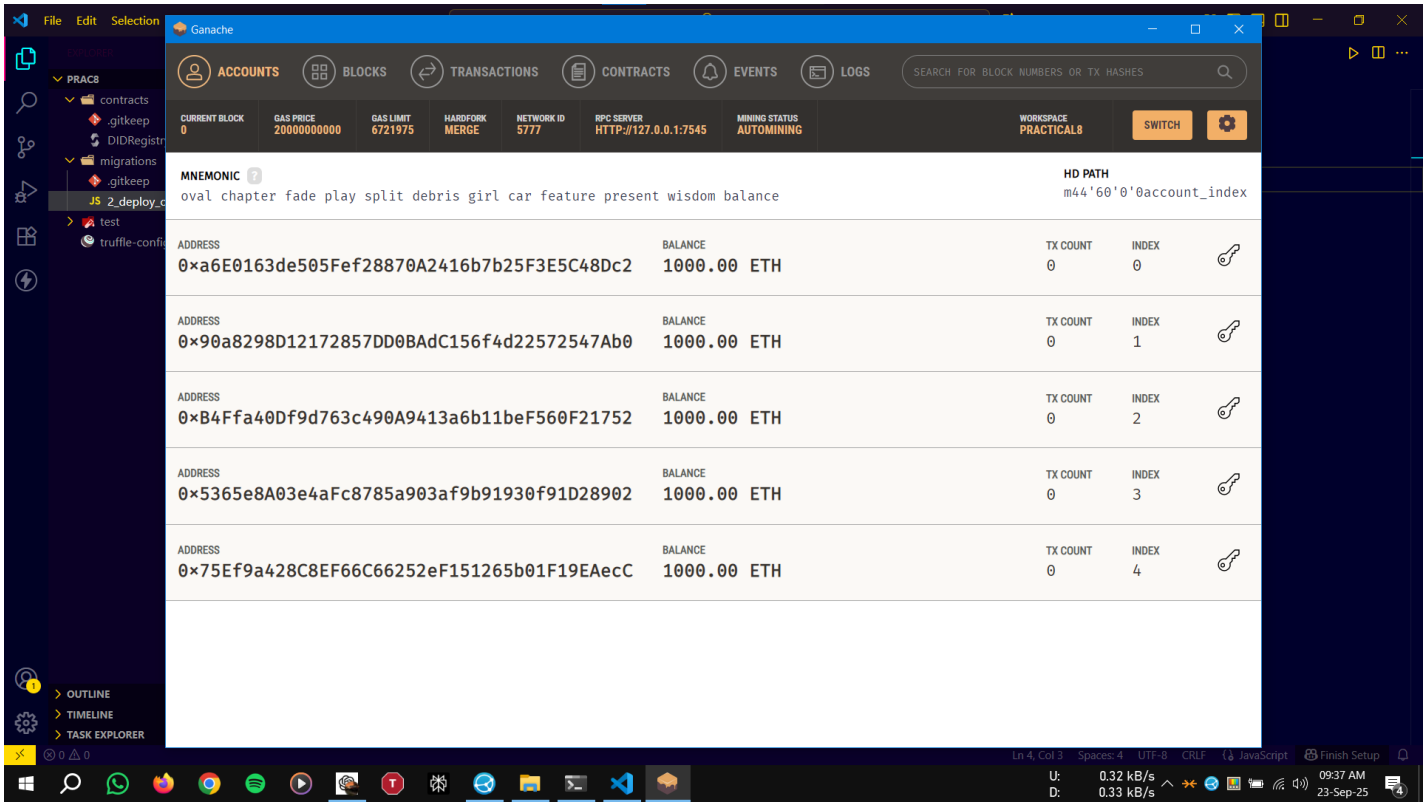
deployer.deploy(DIDRegistry);

};



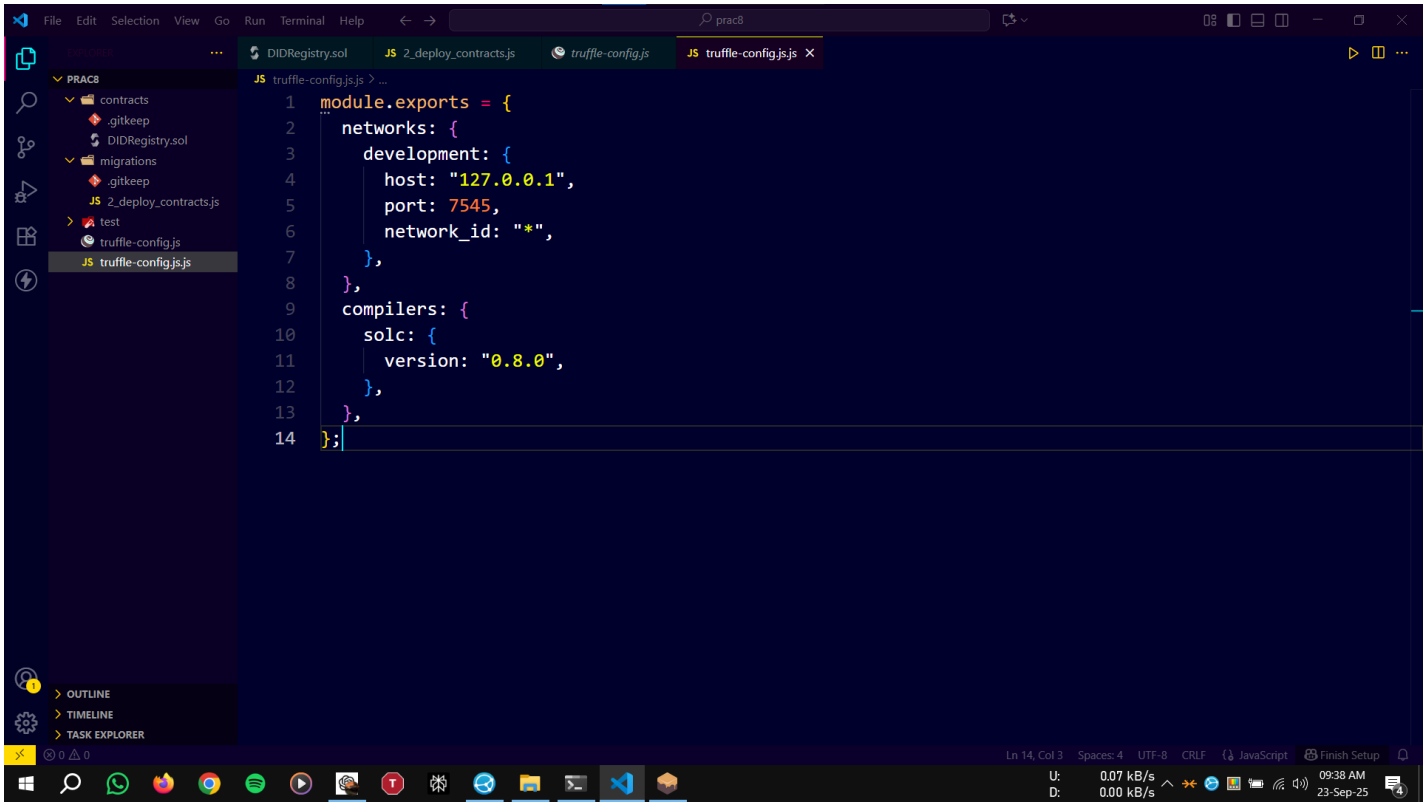
Step3: Deploy

Start Ganache.



Edit truffle-config.js.js according to Ganache host/port.

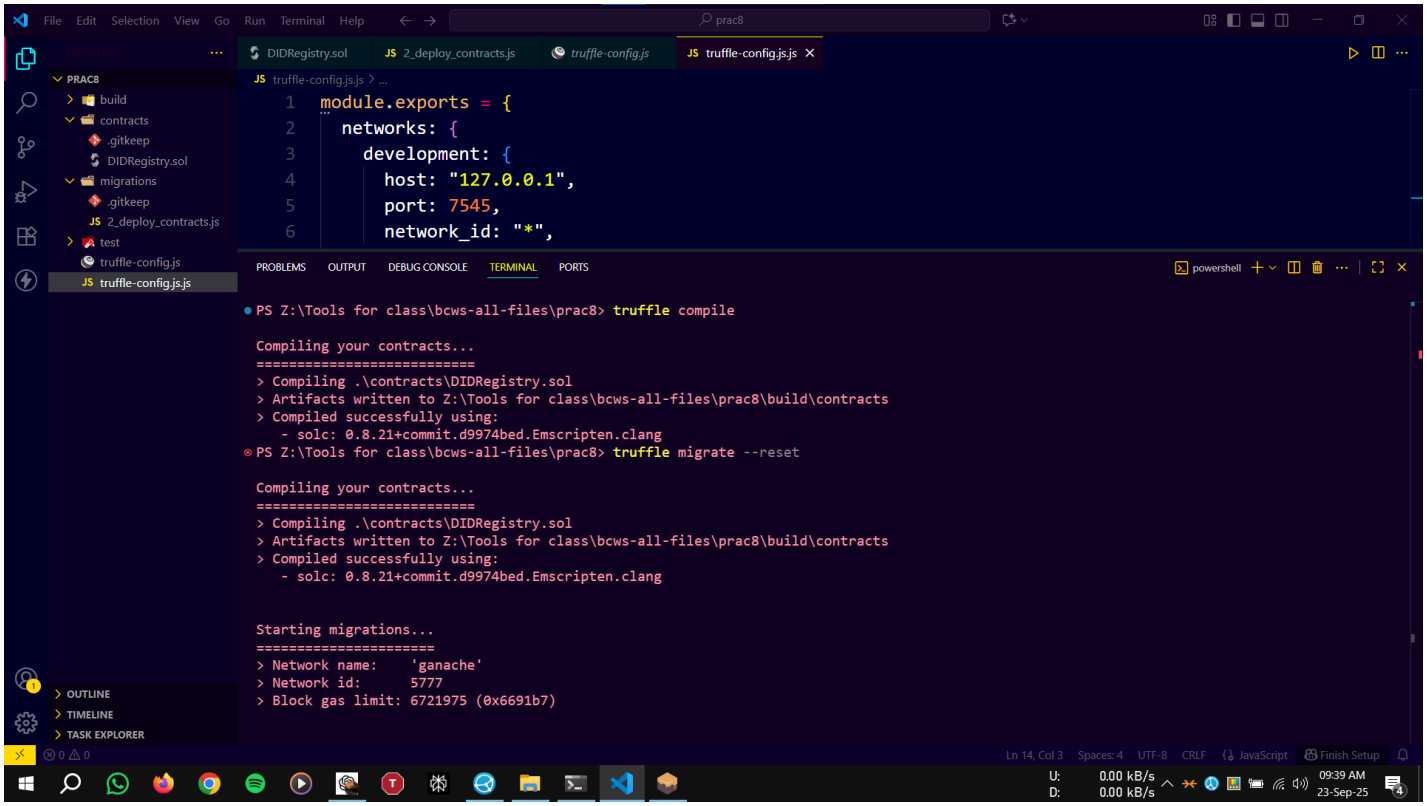
```
module.exports = {
   networks: {
      development: {
         host: "127.0.0.1",
         port: 7545,
         network_id: "*",
      },
   },
   compilers: {
      solc: {
         version: "0.8.0",
      },
   },
};
```



Compile & deploy:

truffle compile

truffle migrate –reset



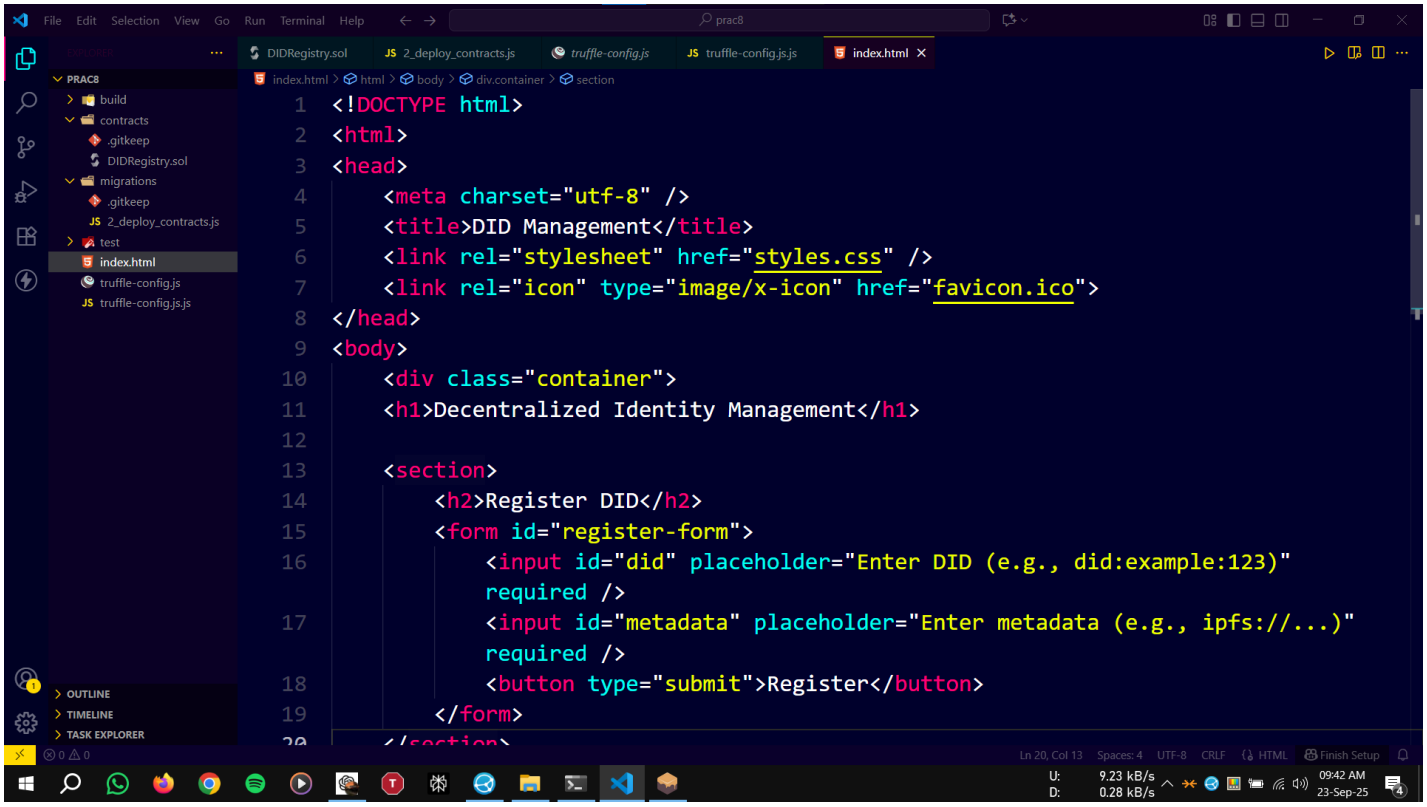
Step4: Frontend

Create 3 files in project root:

index.html (form to register/retrieve DID)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>DID Management</title>
  <link rel="stylesheet" href="styles.css" />
   <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <div class="container">
    <h1>Decentralized Identity Management</h1>

    <section>
      <h2>Register DID</h2>
      <form id="register-form">
        <input id="did" placeholder="Enter DID (e.g., did:example:123)" required />
        <input id="metadata" placeholder="Enter metadata (e.g., ipfs://...)" required />
        <button type="submit">Register</button>
      </form>
    </section>

    <section>
      <h2>Retrieve DID</h2>
      <form id="retrieve-form">
        <input id="retrieveDid" placeholder="Enter DID" required />
        <button type="submit">Retrieve</button>
      </form>
      <div id="did-info"></div>
    </section>
```

```
    <section>
      <h2>Contract Info</h2>
      <p>Contract Address: <span id="http://127.0.0.1:7545"></span></p>
    </section>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

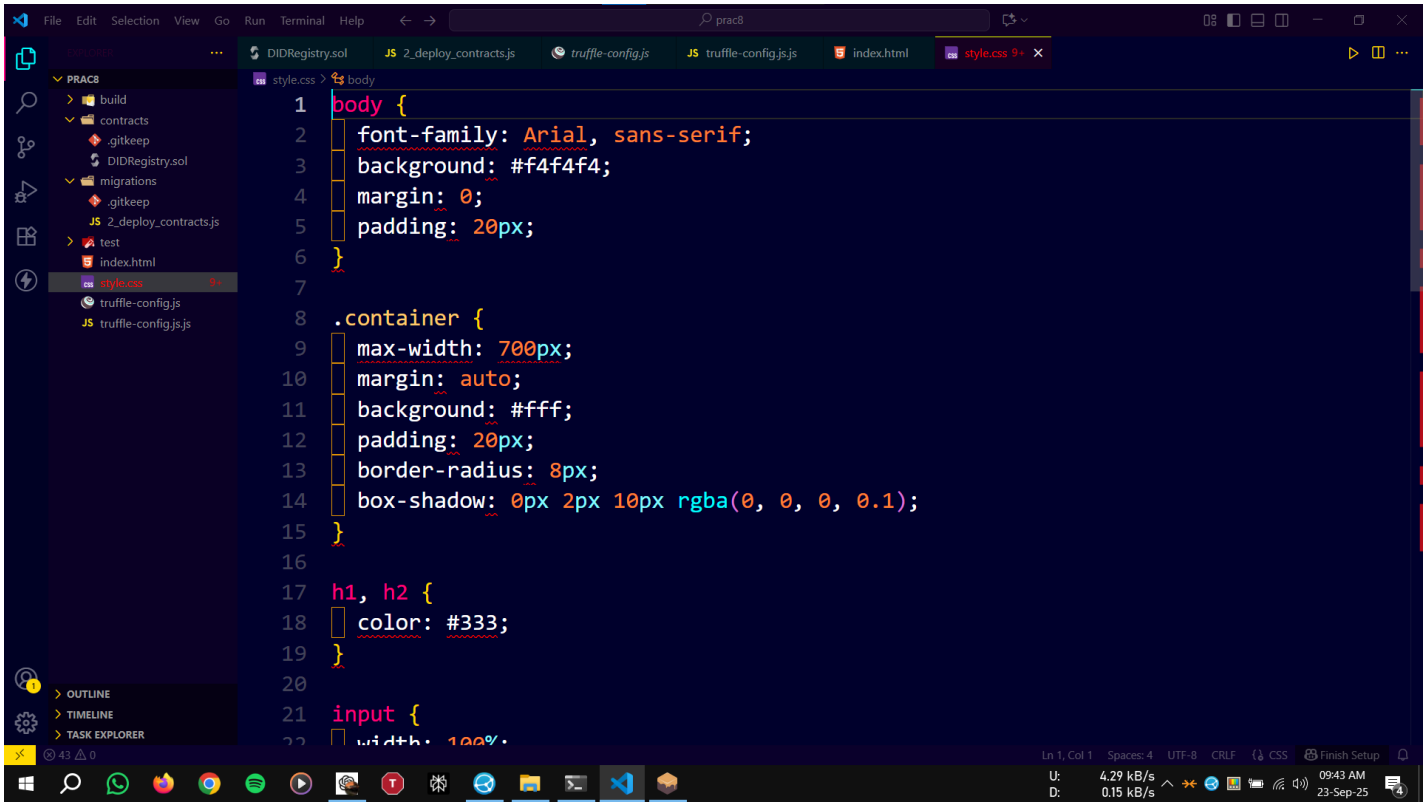


styles.css (basic styling)

```
body {
  font-family: Arial, sans-serif;
  background: #f4f4f4;
  margin: 0;
  padding: 20px;
}
```

```
.container {
  max-width: 700px;
  margin: auto;
  background: #fff;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0px 2px 10px rgba(0, 0, 0, 0.1);
}

h1, h2 {
  color: #333;
}

input {
  width: 100%;
  padding: 10px;
  margin: 6px 0;
  box-sizing: border-box;
  border: 1px solid #ccc;
  border-radius: 4px;
}

button {
  width: 100%;
  padding: 10px;
  background: #007bff;
  color: #fff;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}
```

```
button:hover {
  background: #0056b3;
}

#did-info {
  margin-top: 12px;
  padding: 10px;
  background: #f1f1f1;
  border-radius: 4px;
}
```

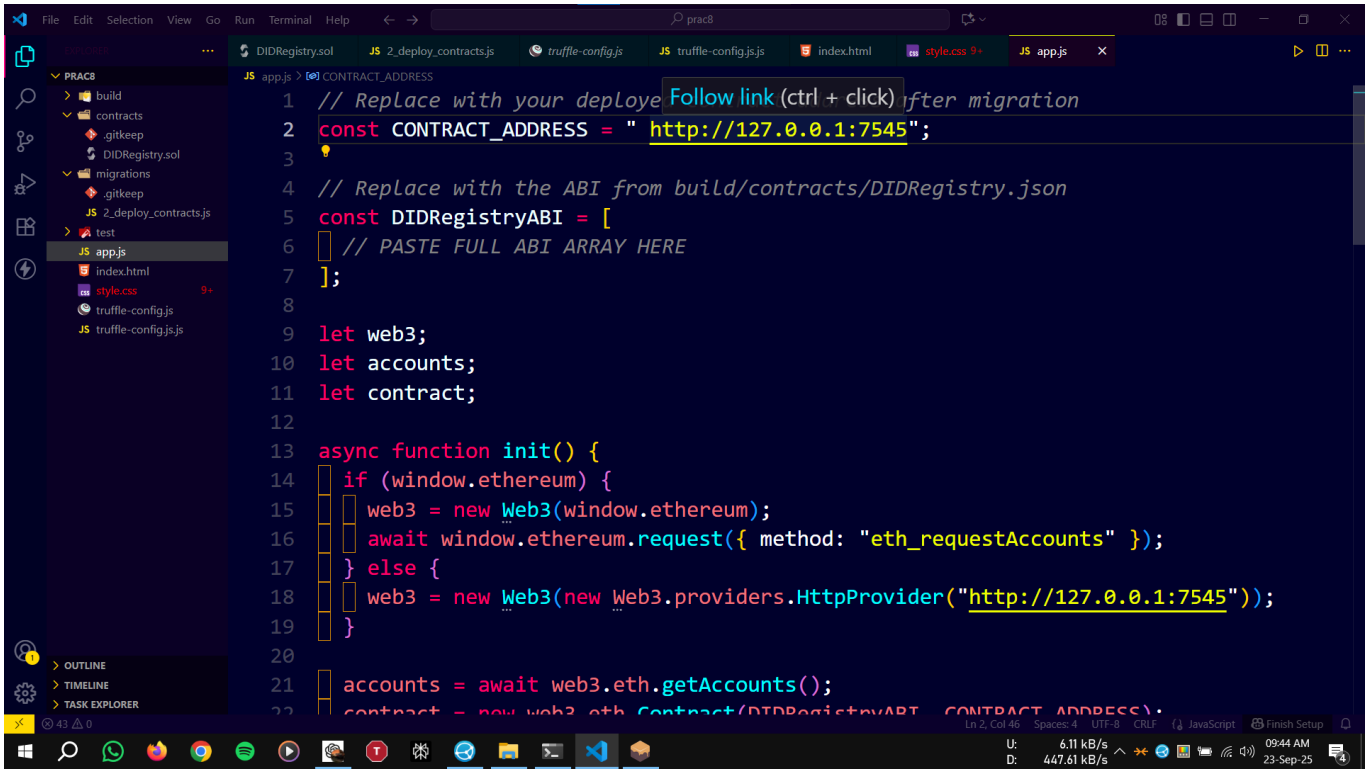


app.js (connect to contract with Web3.js)

```
// Replace with your deployed contract address after migration
const CONTRACT_ADDRESS = " http://127.0.0.1:8545";

// Replace with the ABI from build/contracts/DIDRegistry.json
const DIDRegistryABI = [
  // PASTE FULL ABI ARRAY HERE
```

```
];

let web3;
let accounts;
let contract;

async function init() {
  if (window.ethereum) {
    web3 = new Web3(window.ethereum);
    await window.ethereum.request({ method: "eth_requestAccounts" });
  } else {
    web3 = new Web3(new Web3.providers.HttpProvider("http://127.0.0.1:7545"));
  }

  accounts = await web3.eth.getAccounts();
  contract = new web3.eth.Contract(DIDRegistryABI, CONTRACT_ADDRESS);

  document.getElementById("contract-address").innerText = CONTRACT_ADDRESS;
}

document.getElementById("register-form").onsubmit = async (e) => {
  e.preventDefault();
  const did = document.getElementById("did").value.trim();
  const metadata = document.getElementById("metadata").value.trim();

  try {
    await contract.methods.registerDID(did, metadata).send({ from: accounts[0] });
    alert("DID registered successfully!");
  } catch (err) {
    console.error(err);
    alert("Registration failed: " + err.message);
  }
```

```
};

document.getElementById("retrieve-form").onsubmit = async (e) => {
  e.preventDefault();
  const did = document.getElementById("retrieveDid").value.trim();

  try {
    const result = await contract.methods.getDID(did).call();
    const controller = result[0];
    const metadata = result[1];
    const updatedAt = result[2];
    const time = updatedAt > 0 ? new Date(updatedAt * 1000).toLocaleString() : "Never";

    document.getElementById("did-info").innerHTML = `
      <p><strong>DID:</strong> ${did}</p>
      <p><strong>Controller:</strong> ${controller}</p>
      <p><strong>Metadata:</strong> ${metadata}</p>
      <p><strong>Updated At:</strong> ${time}</p>
    `;
  } catch (err) {
    console.error(err);
    alert("Retrieval failed: " + err.message);
  }
};
window.addEventListener("load", init);
```



Add the address of smart contract into app.js obtained from truffle migrate command from the terminal

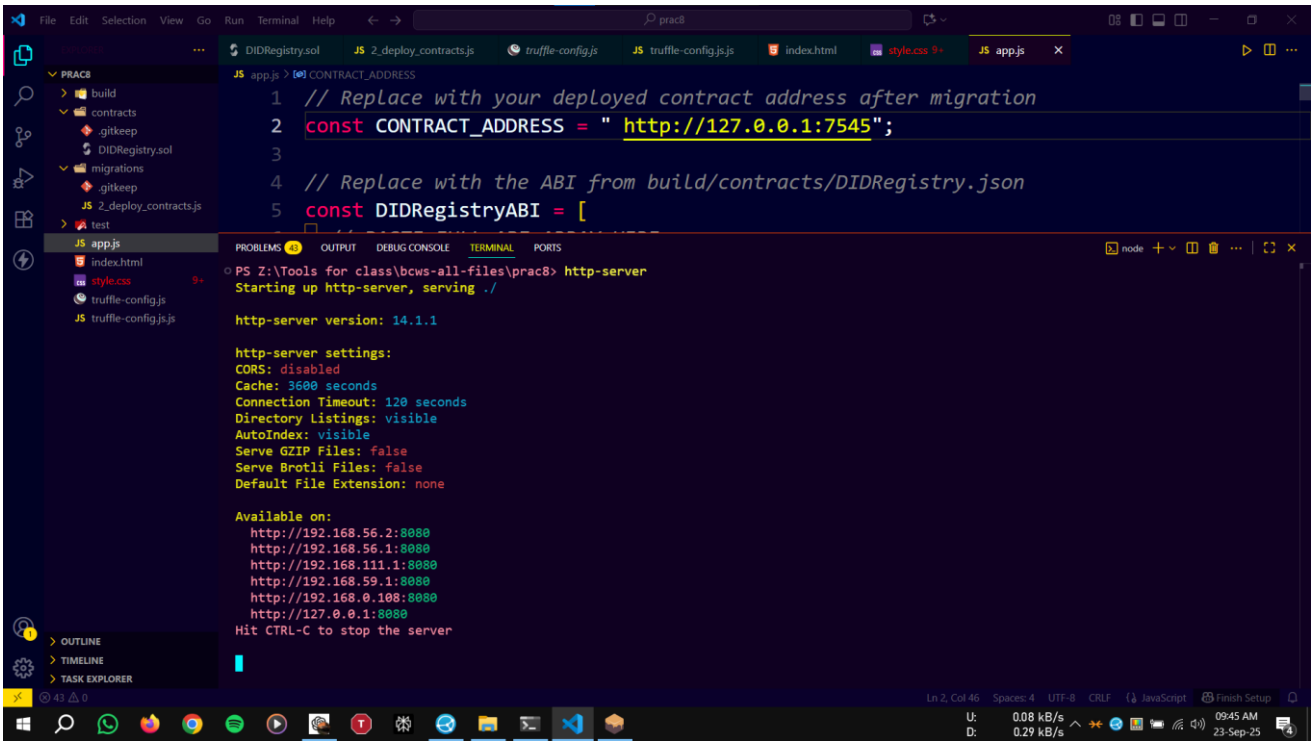
Step5: Run App

Start local server:

http-server

Open http://127.0.0.1:8080 in browser.

Connect MetaMask → point it to Ganache network.

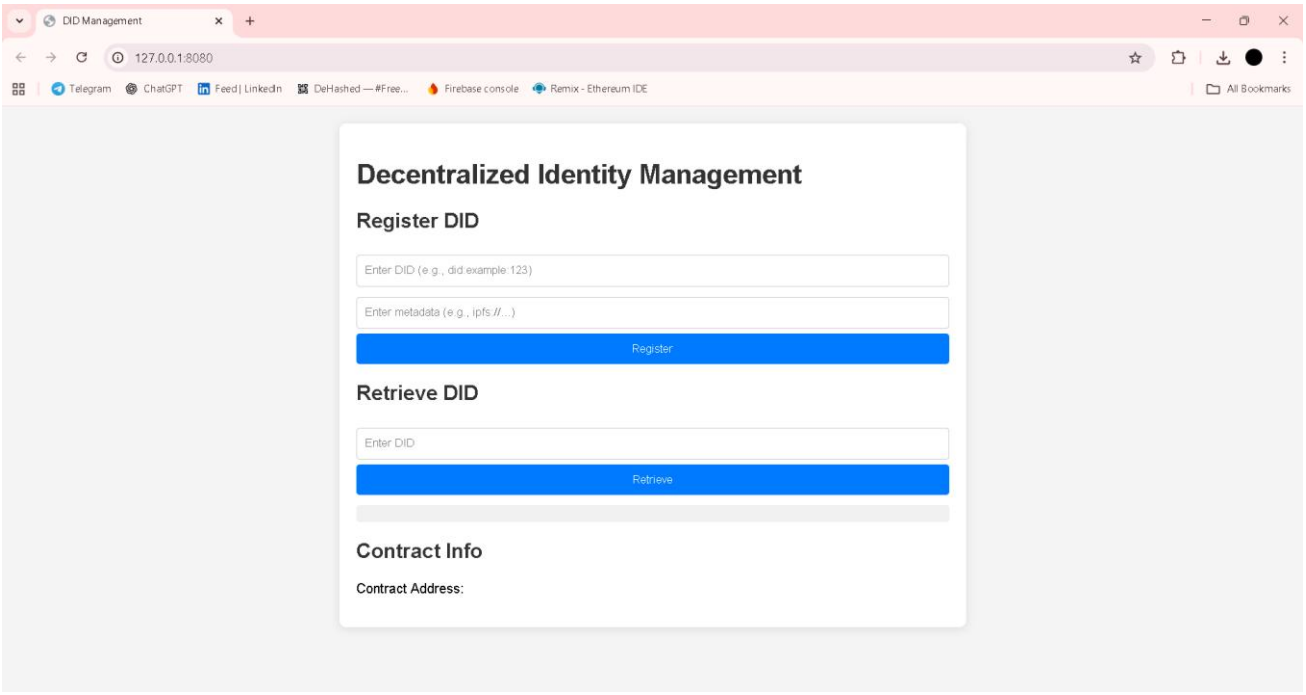


Step6: Test

Use form to register a DID.

Use form to retrieve DID info.

Confirm success messages and on-chain data.



Practical - 9

Aim: Consensus Mechanisms and Security Implications

1. Understanding various consensus algorithms
2. Analyzing security implications of different consensus mechanisms
3. Hands-on exploration of consensus protocols

Theory -

In decentralized networks and blockchain systems, consensus mechanisms ensure that all participating nodes agree on a common state of the ledger without relying on a central authority. They are crucial for maintaining integrity, preventing fraud, and ensuring network security.

1. Consensus Algorithms

1.1 Proof of Work (PoW)

- **Mechanism:** Miners solve computational puzzles to validate transactions and produce new blocks.
- **Use Case:** Bitcoin and early cryptocurrencies.
- **Advantages:** Highly secure, decentralized, resistant to Sybil attacks.
- **Drawbacks:** Energy-intensive, low throughput (e.g., Bitcoin ~7 TPS).

1.2 Proof of Stake (PoS)

- **Mechanism:** Validators are chosen based on the tokens they lock as collateral.
- **Use Case:** Ethereum 2.0.
- **Advantages:** Energy-efficient, faster than PoW.
- **Drawbacks:** Risk of centralization; wealthy validators gain more influence.

1.3 Delegated Proof of Stake (DPoS)

- **Mechanism:** Token holders elect a small set of delegates to validate transactions.
- **Use Case:** EOS, Tron, Lisk.
- **Advantages:** High scalability and performance.
- **Drawbacks:** Risk of centralization and vulnerability to targeted DDoS attacks.

1.4 Practical Byzantine Fault Tolerance (PBFT)

- **Mechanism:** Nodes (replicas) achieve consensus by majority agreement, tolerating faulty or malicious actors.
- **Use Case:** Hyperledger Fabric.
- **Advantages:** Low transaction finality time, tolerant to node failures.
- **Drawbacks:** Communication overhead, scalability issues in large networks.

1.5 Proof of Authority (PoA)

- **Mechanism:** A limited number of trusted authorities validate blocks.
- **Use Case:** Private/consortium blockchains (e.g., VeChain).
- **Advantages:** Efficient, fast, suitable for permissioned systems.
- **Drawbacks:** Centralized by design, requires trust in validators.

2. Security Implications

2.1 Proof of Work (PoW)

- **51% Attack:** Possible if majority hash power is controlled by one entity.
- **Sybil Resistance:** High, due to cost of obtaining hash power.
- **Issue:** Enormous energy consumption; potential environmental and regulatory concerns.

2.2 Proof of Stake (PoS)

- **Staking Security:** Malicious validators risk losing funds (slashing).
- **Nothing-at-Stake Problem:** Validators may vote on multiple chains.
- **Risk:** Wealthy validators gain more power, reducing fairness.

2.3 Delegated Proof of Stake (DPoS)

- **Centralization:** Small set of delegates may monopolize control.
- **Vulnerability:** DDoS attacks against few validators can disrupt the network.

2.4 Byzantine Fault Tolerance (BFT) Protocols

- **Strength:** Tolerant as long as <1/3 of nodes are faulty.
- **Challenges:** High communication overhead and poor scalability in large public chains.

2.5 Proof of Authority (PoA)

- **Trust Model:** Relies on trusted authorities; compromised nodes can harm the system.
- **Centralization:** Security trade-off for efficiency.

3. Hands-on Exploration of Consensus Protocols

To better understand consensus mechanisms, we simulate two widely used protocols: Proof of Work (PoW) and Proof of Stake (PoS).

3.1 Proof of Work (PoW) Exercise

- **Tools:** Bitcoin Testnet, Mining Simulator (e.g., CGMiner, BFGMiner).
- **Steps:**
    1. Configure and connect a node on Bitcoin Testnet.
    2. Start mining blocks and track hash rate and energy usage.
    3. Attempt a double-spend attack in the test environment.

- **Observations:**
  - Mining difficulty adjusts dynamically.
  - A 51% attack is highly impractical due to cost and computational requirements.

3.2 Proof of Stake (PoS) Exercise

- **Tools:** Ethereum 2.0 Testnet (Prysm, Lighthouse).
- **Steps:**
  1. Set up a validator node on Ethereum 2.0 testnet.
  2. Stake ETH (testnet tokens) to participate in validation.
  3. Observe reward distribution and penalties for downtime.
  4. Attempt malicious behavior and observe slashing.
- **Observations:**
  - Energy requirements are minimal compared to PoW.
  - Validators must maintain consistent uptime to avoid penalties.

Practical - 10

Aim: Blockchain Security Audit and Best Practices

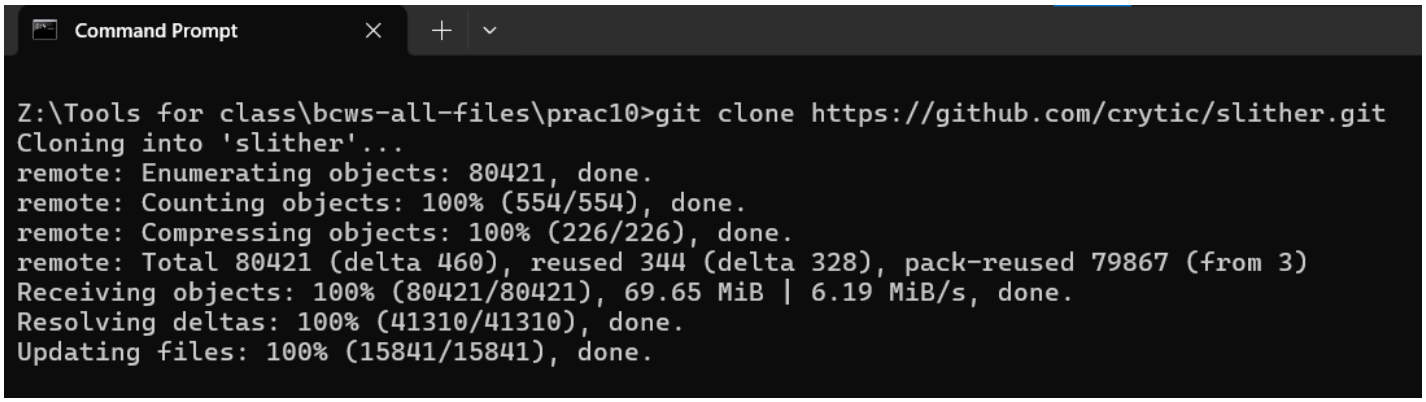
1. Conducting a security audit for a blockchain application
2. Implementing advanced security measures and best practices
3. Presentation and discussion of security findings

Tools Required -

1. Blockchain Development Framework: Truffle.
2. Local Blockchain Network: Ganache.
3. Smart Contract Development: Solidity.
4. Development Environment: VS Code.
5. Audit Tools: MythX, Slither.
6. Web Interface: HTML/CSS/JavaScript with MetaMask

Step1: Download slither from https://github.com/crytic/slither?tab=readme-ov-file#how-to-install

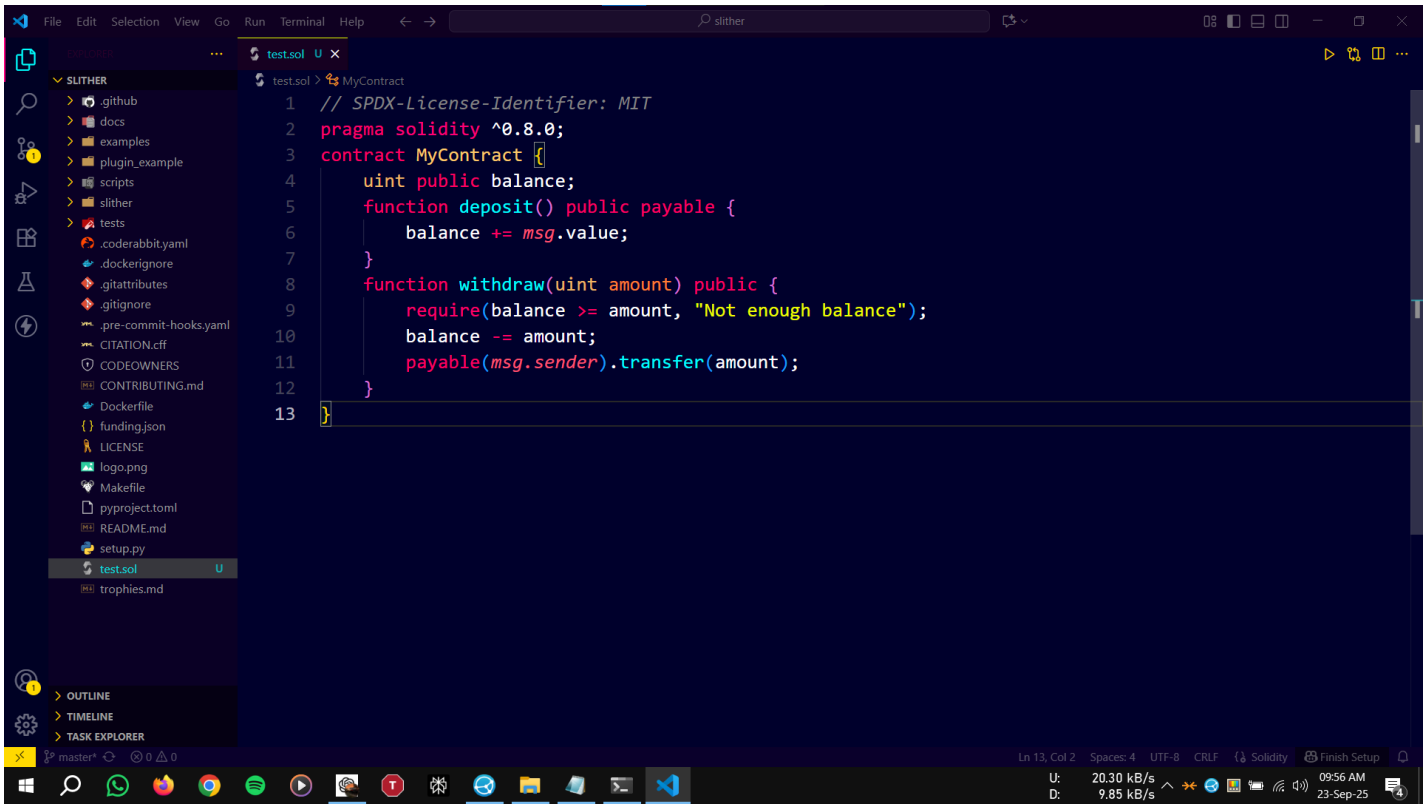
git clone https://github.com/crytic/slither.git



Step2: In a slither folder create test.sol file with the vulnerable code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MyContract {
    uint public balance;
    function deposit() public payable {
        balance += msg.value;
    }
```

```
    function withdraw(uint amount) public {
        require(balance >= amount, "Not enough balance");
        balance -= amount;
        payable(msg.sender).transfer(amount);
    }
}
```



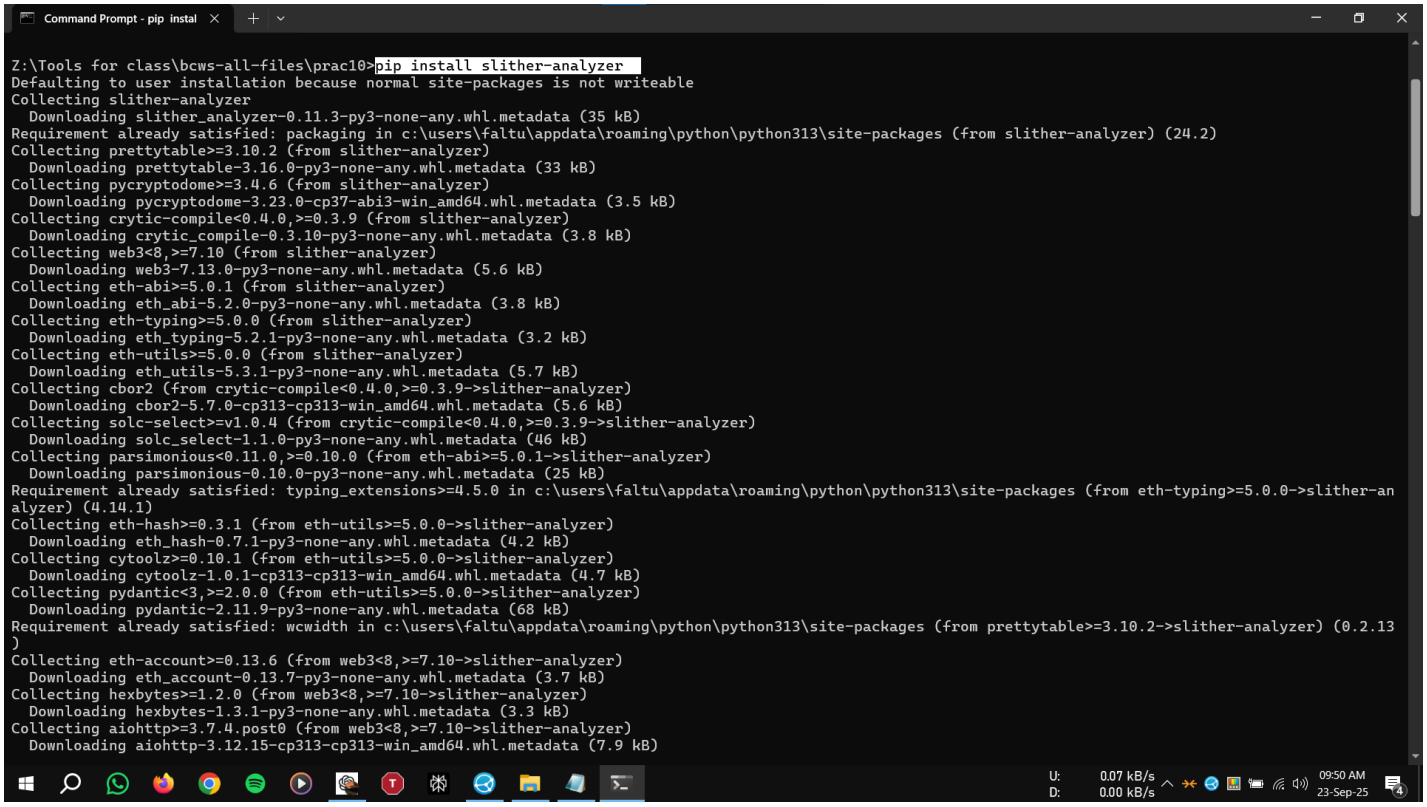
Step3: Slither is a Python package, so you need Python + pip.

Make sure you have Python 3.8+ installed:

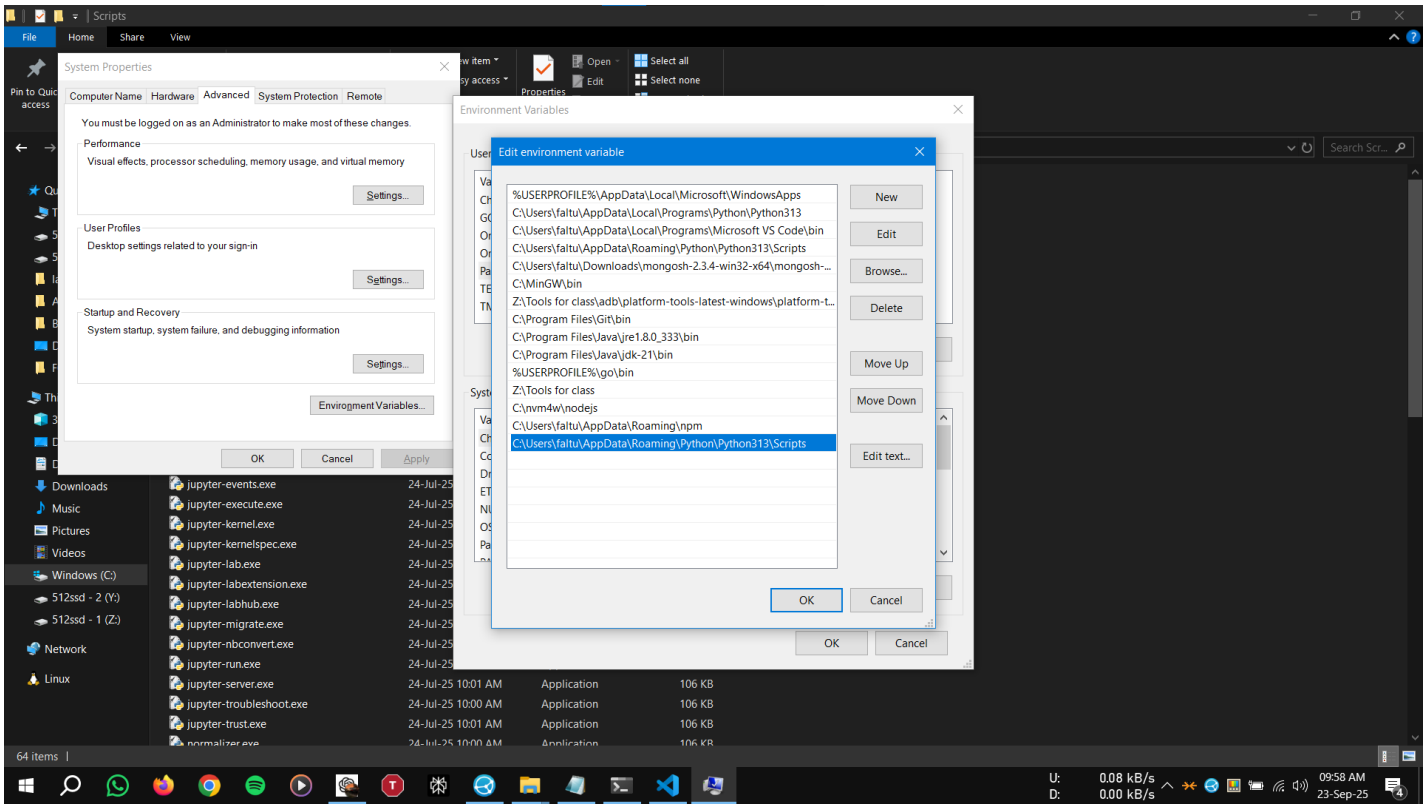
python --version

Install Slither globally:

pip install slither-analyzer

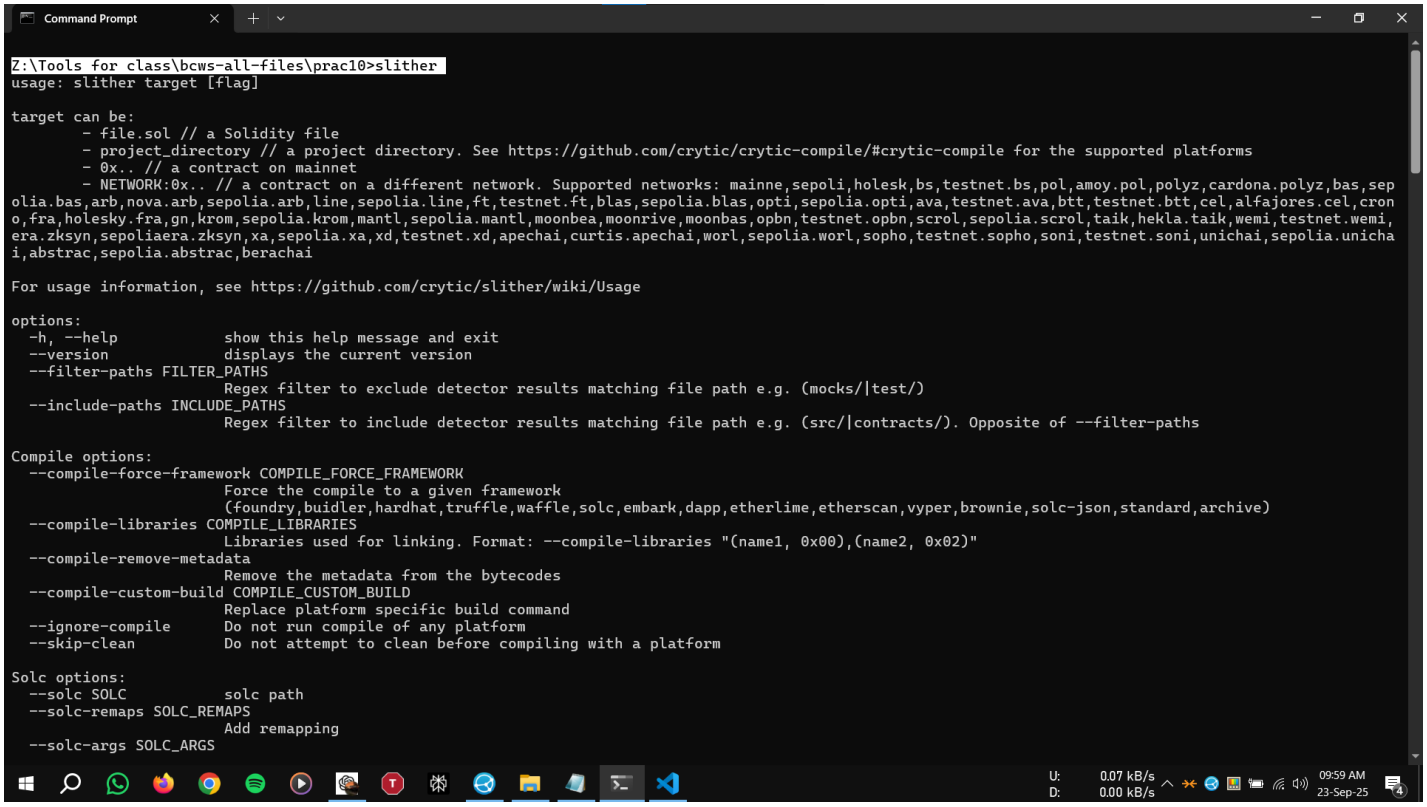


Step4: Open environment variable & add the Scripts folder to PATH.



Step5: Open cmd and run the command:

slither



Step6: Run the command given below to detect the vulnerabilities in the contract and also give the detail info about the vulnerabilities:

slither test.sol