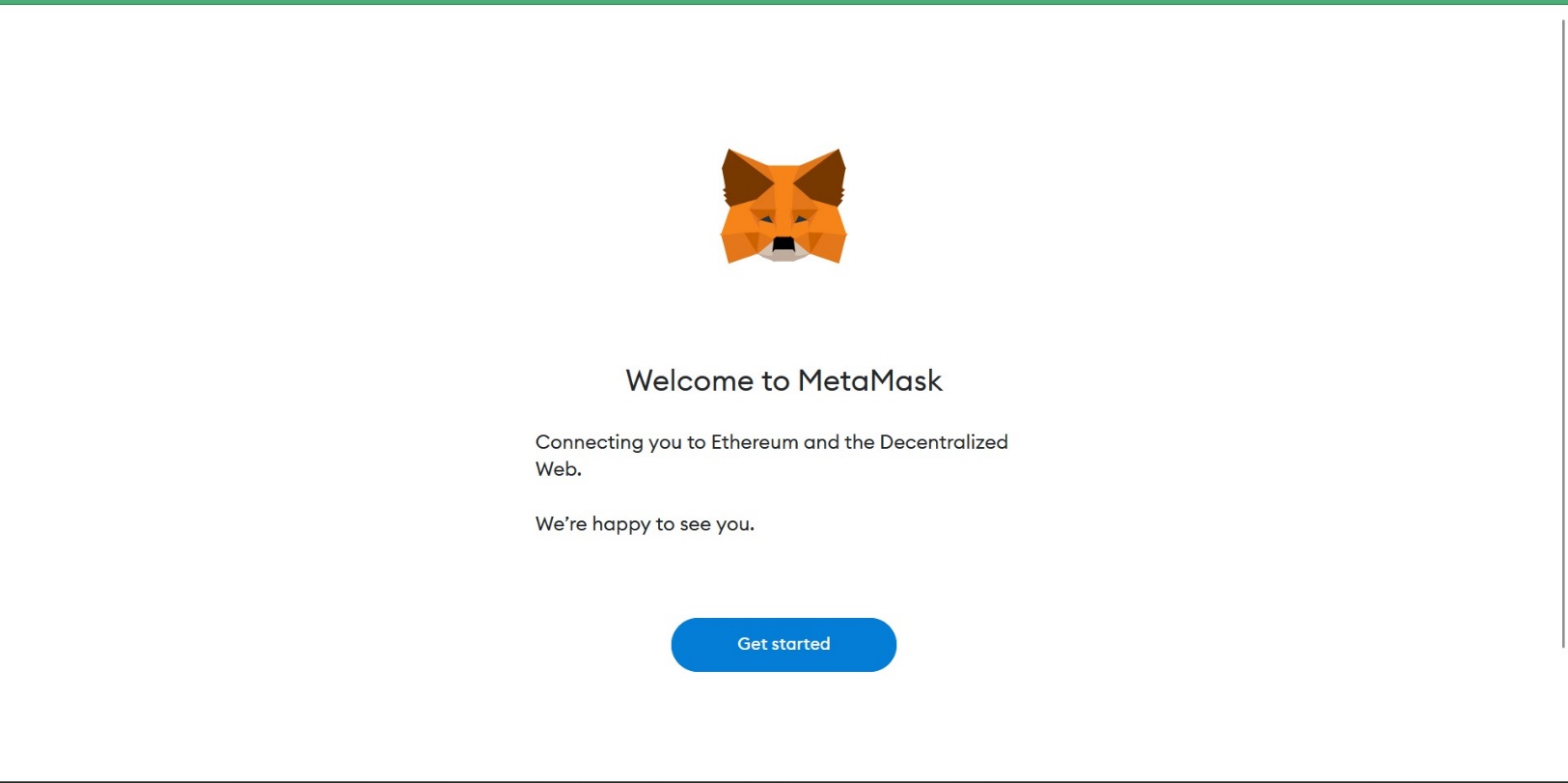
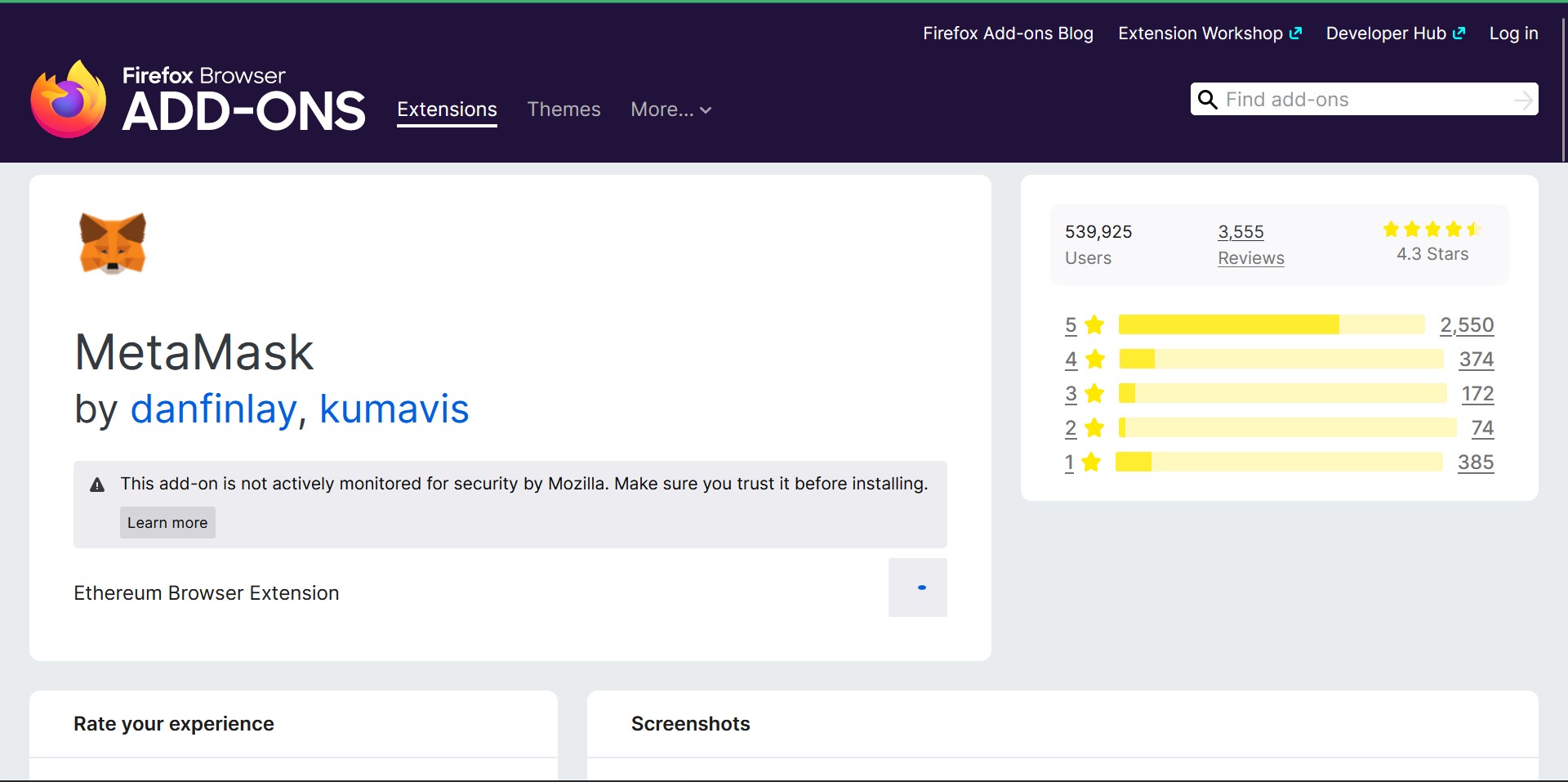
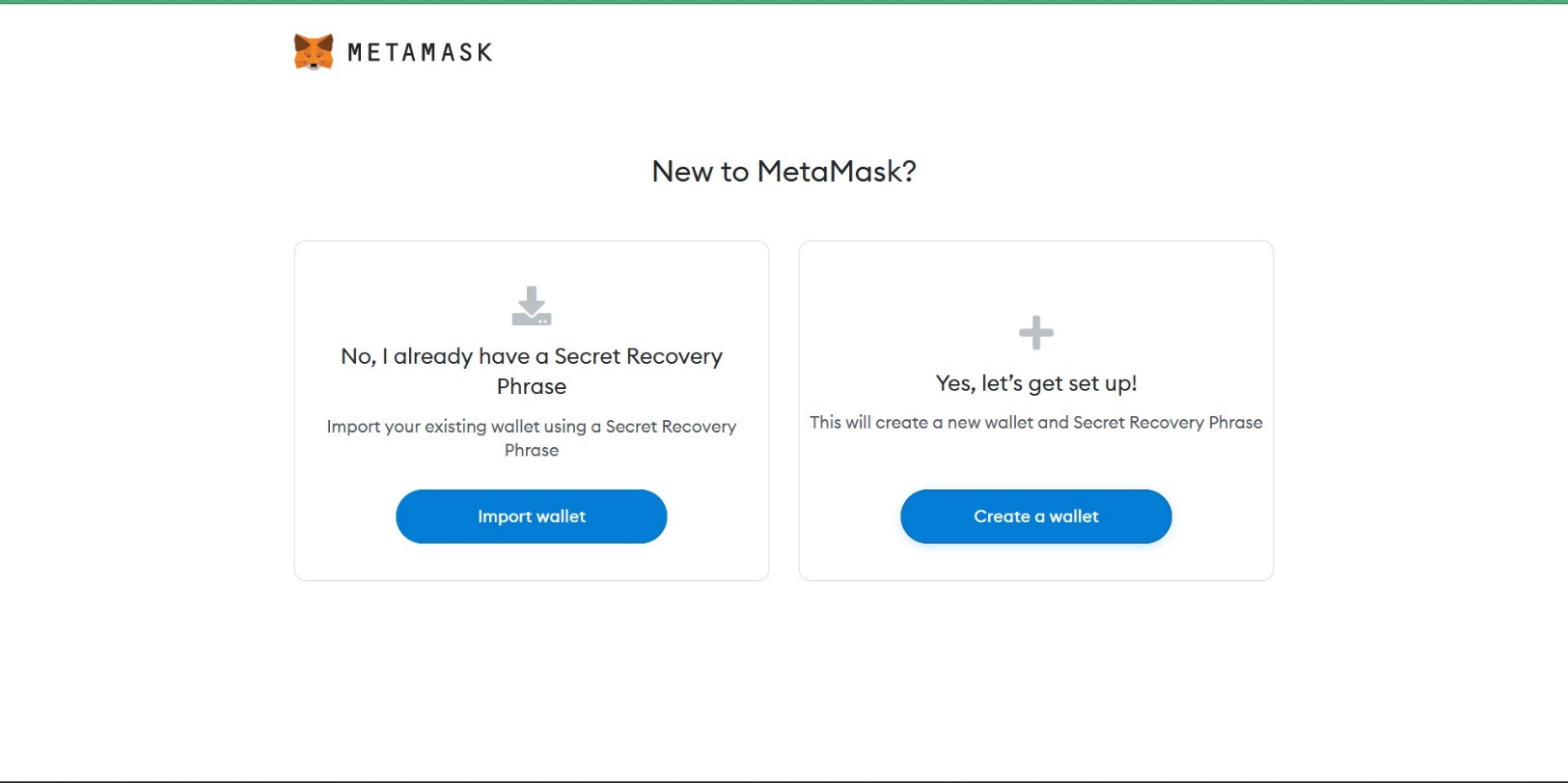
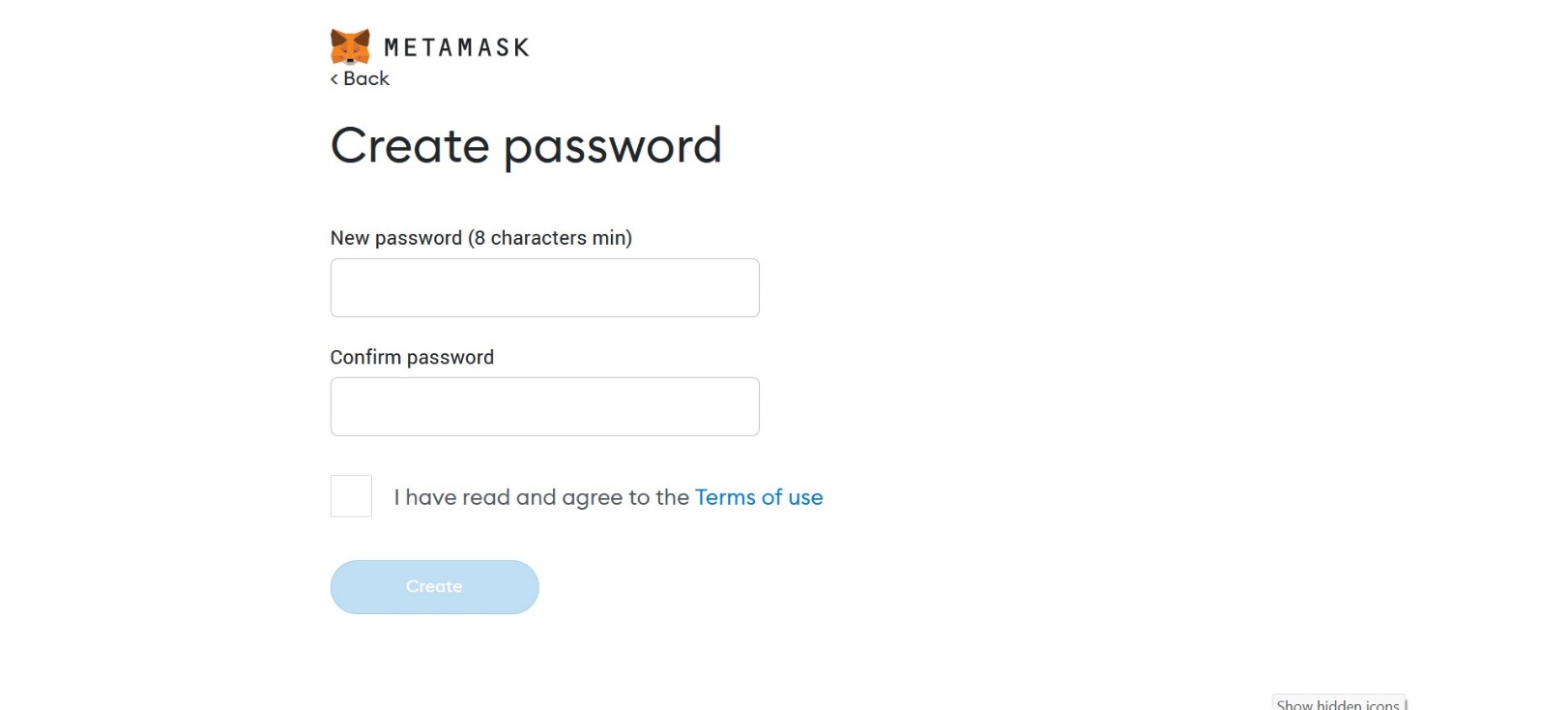
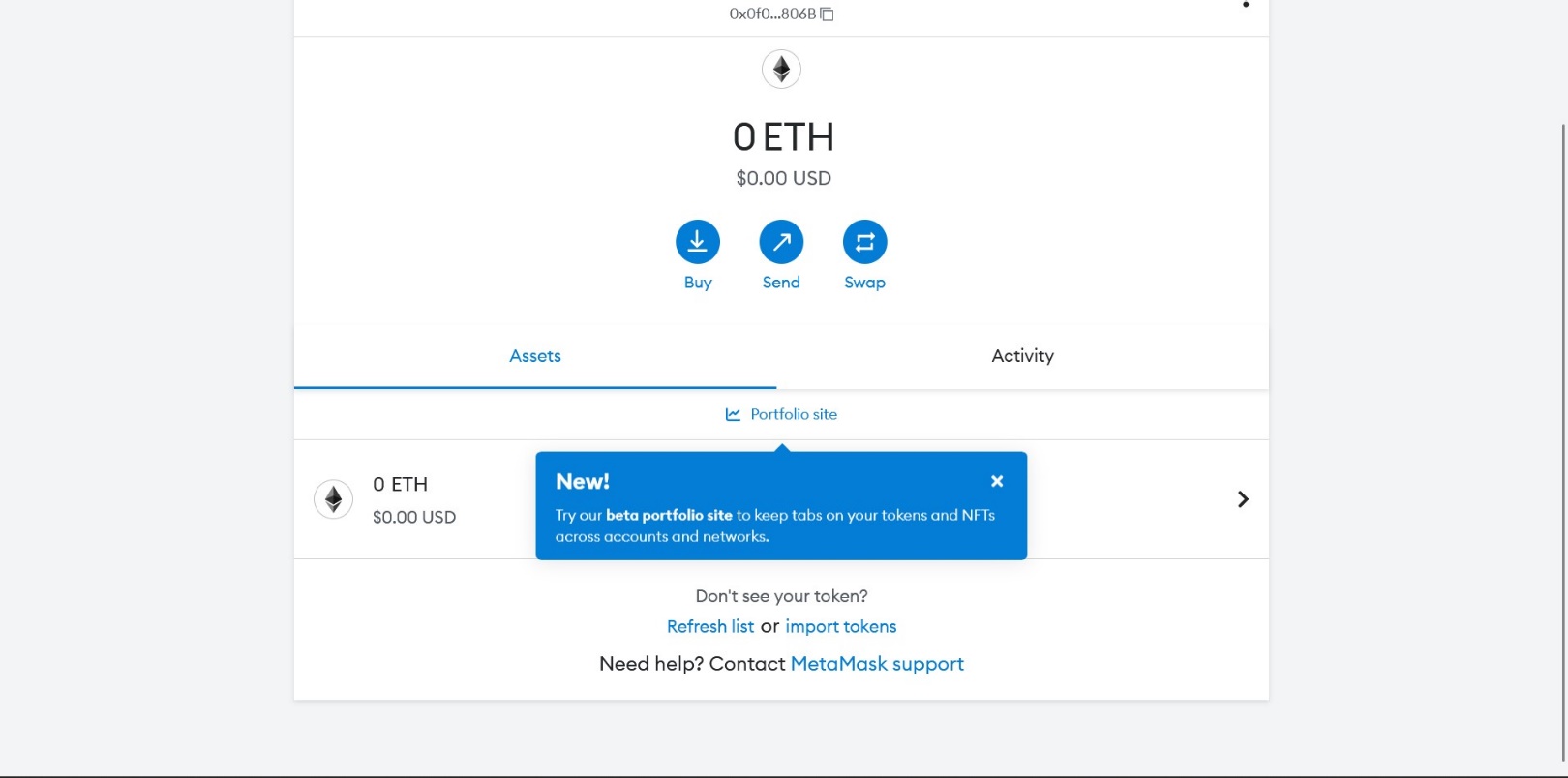
Assignment no. 1

Installation of Metamask and Study spending Ether per transaction

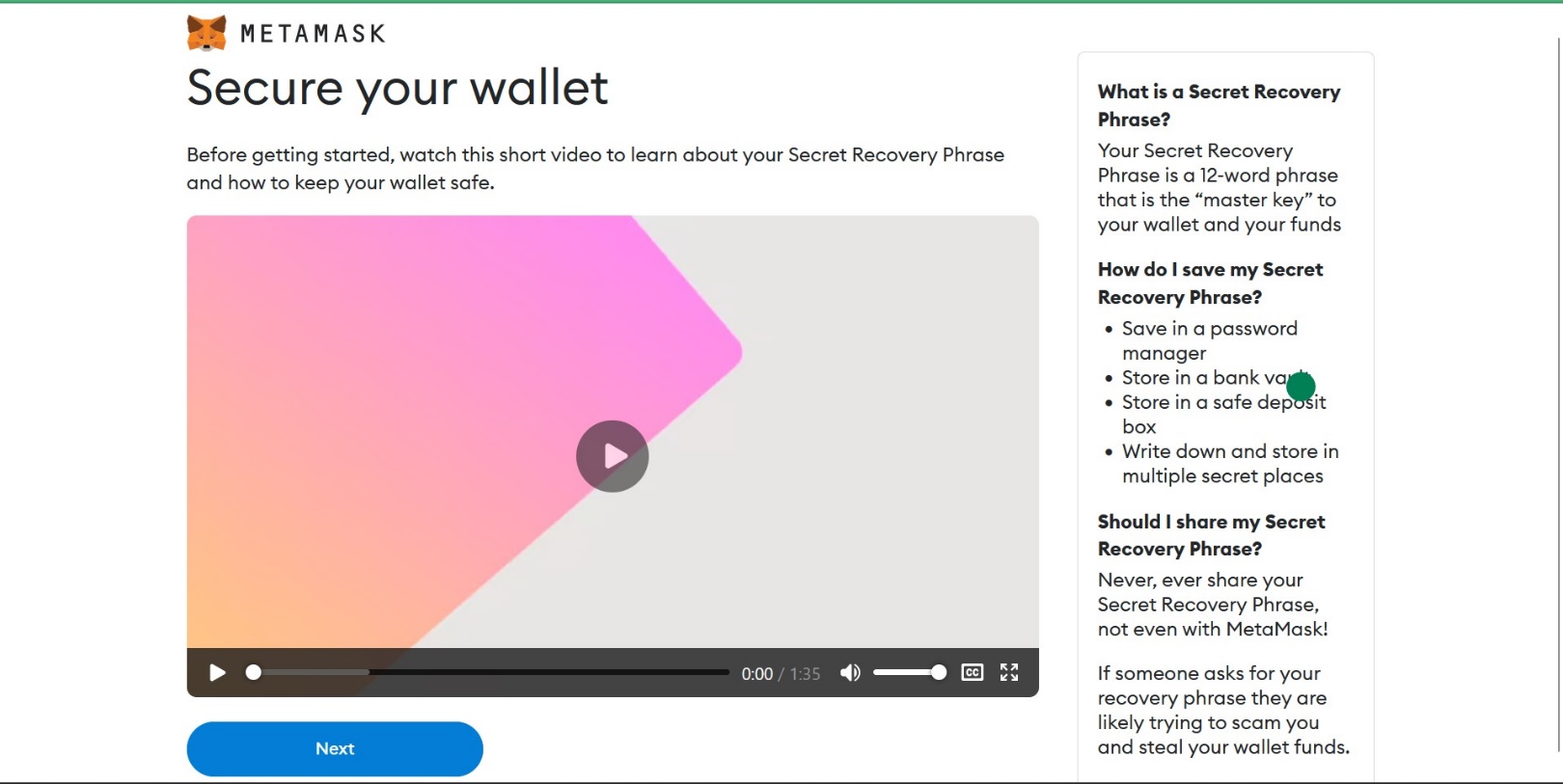


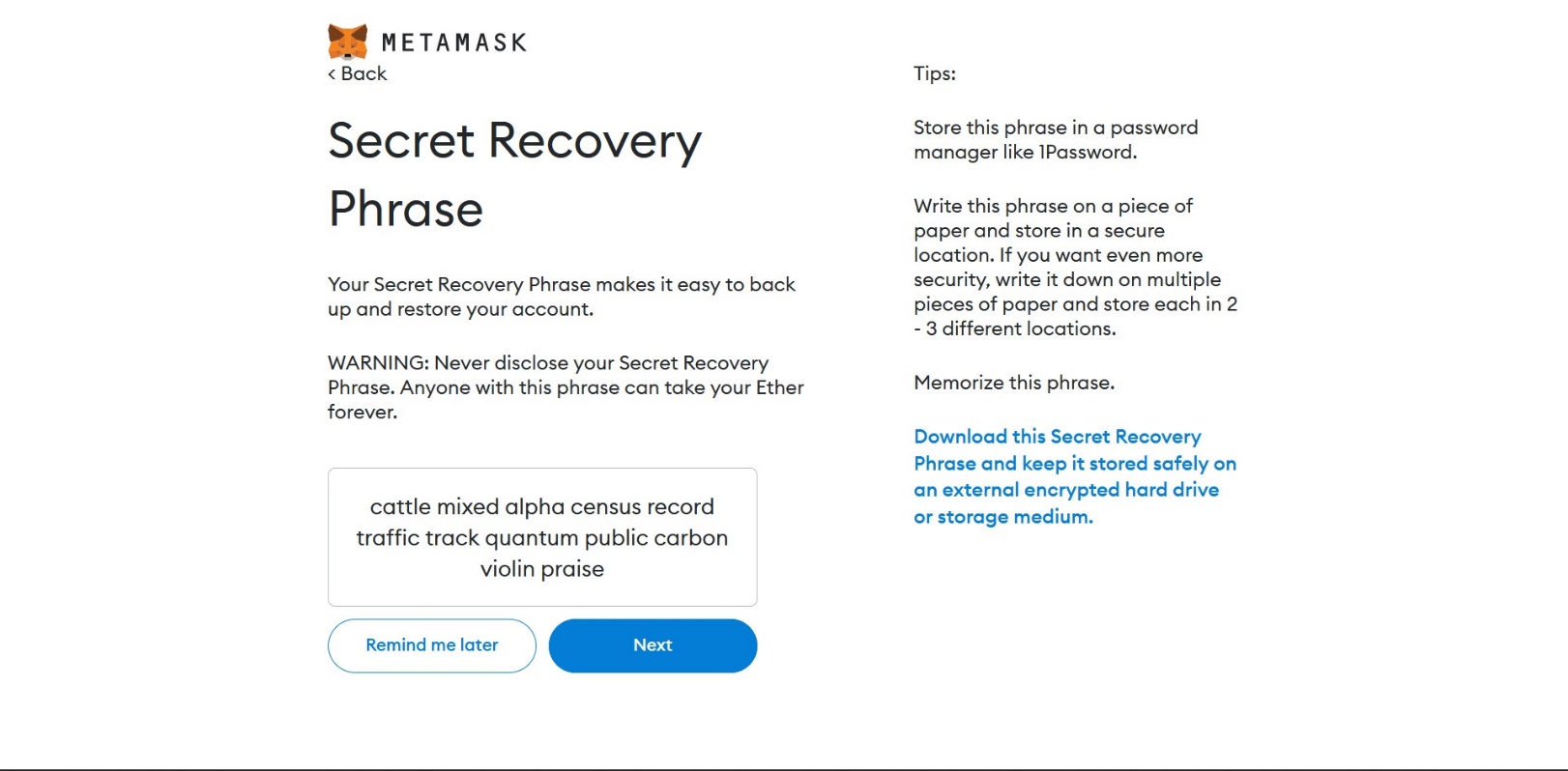
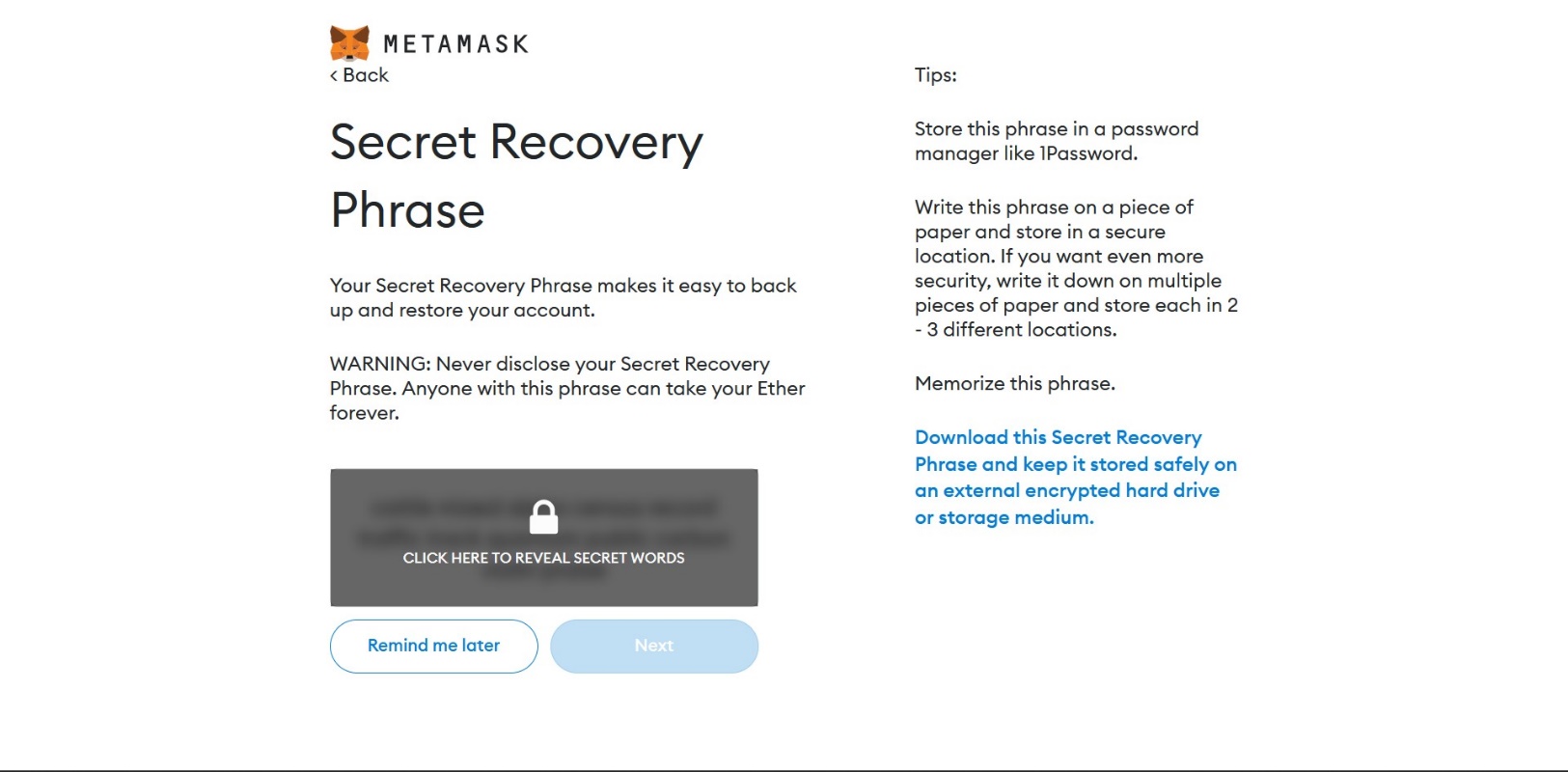


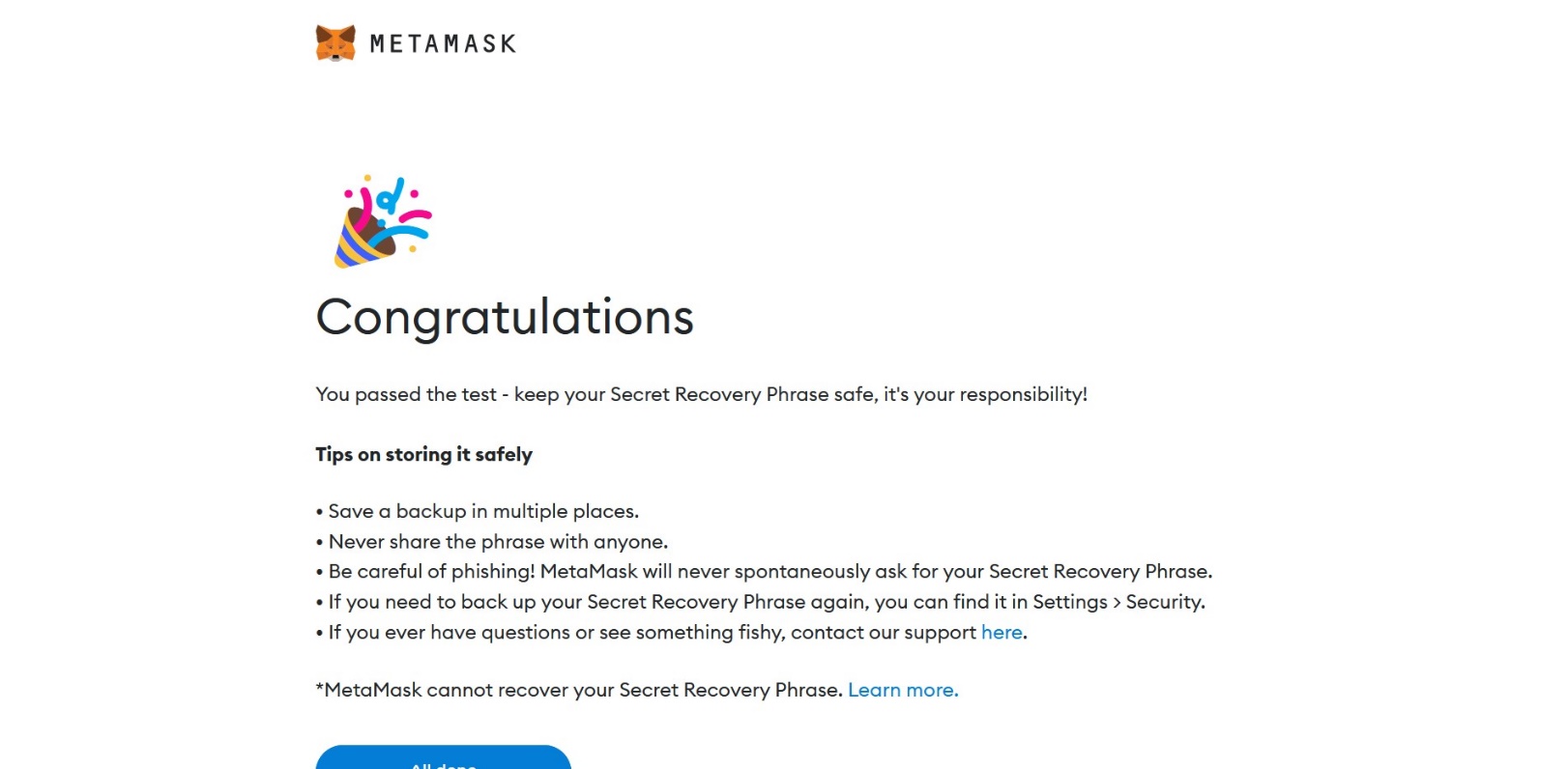


Assignment no. 2

Create Your Own Wallet using Metamask for Crypto Transactions







Assignment no. 3

Write a Smart Contract on a Test Network for Bank Account of a Customer.

Code:

pragma solidity ^0.6;

// Write  a  smart  contract  on  a  test  network,  for  Bank  account  of  a  customer  for  // following operations: Deposit money | Withdraw Money | Show balance

// SPDX-License-Identifier: MIT

contract banking{

    mapping(address => uint) public user\_account;

    mapping(address=>bool) public user\_exists;

    function create\_account() public payable returns (string memory){

        require(user\_exists[msg.sender] == false, 'Account already created');

        if(msg.value == 0){

            user\_account[msg.sender] = 0;

            user\_exists[msg.sender] = true;

            return "Account Created";

        }

        require(user\_exists[msg.sender] == false, "Account already created");

        user\_account[msg.sender] = msg.value;

        user\_exists[msg.sender] = true;

        return "Account created";

    }

    function deposit() public payable returns (string memory){

        require(user\_exists[msg.sender] == true, "Account not created");

        require(msg.value > 0, "Value for deposit is Zero");

        user\_account[msg.sender] = user\_account[msg.sender] + msg.value;

        return "Deposisted Successfully";

    }

    function withdraw(uint amount) public payable returns (string memory){

        require(user\_account[msg.sender] > amount, "Insufficient Balance");

        require(user\_exists[msg.sender] == true, "Account not created");

        require(msg.value > 0, "Value for withdraw is Zero");

        user\_account[msg.sender] = user\_account[msg.sender] - amount;

        payable(msg.sender).transfer(amount);

        return "Withdrawn Successfully";

    }

    function transfer(address payable userAddress, uint amount) public returns(string memory){

        require(user\_account[msg.sender] > amount, "Insufficient Balance");

        require(user\_exists[msg.sender] == true, "Account not created");

        require(user\_exists[userAddress] == true, "Account not created");

        require(amount > 0, "Amount should be greater than 0");

        user\_account[msg.sender] = user\_account[msg.sender] - amount;

        user\_account[userAddress] = user\_account[userAddress] + amount;

        return "Transfered Successfully";

    }

    function send\_amt(address payable toAddress, uint256 amount) public payable returns(string memory){

        require(user\_account[msg.sender] > amount, "Insufficient Balance");

        require(user\_exists[msg.sender] == true, "Account not created");

        require(amount > 0, "Amount should be greater than zero");

        user\_account[msg.sender] = user\_account[msg.sender] - amount;

        toAddress.transfer(amount);

        return "Transfer Successfull";

    }

    function user\_balance() public view returns(uint){

        return user\_account[msg.sender];

    }

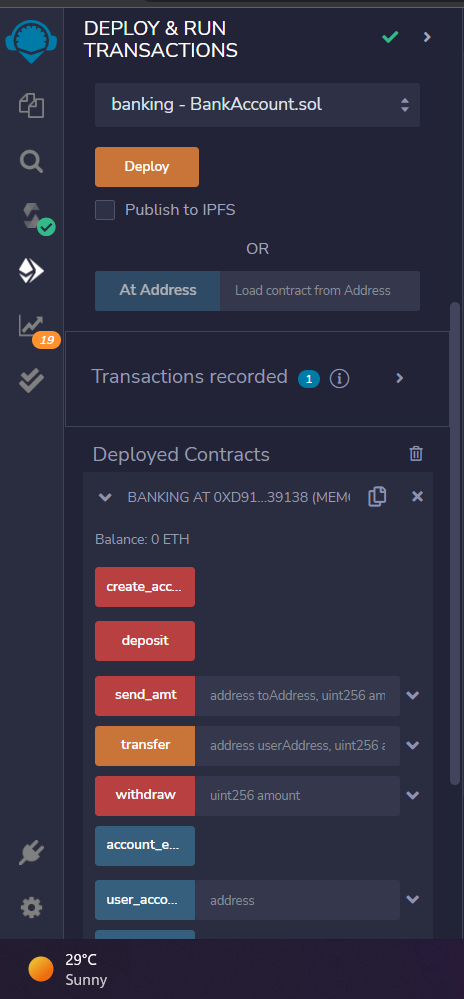
    function account\_exists() public view returns(bool){

        return user\_exists[msg.sender];

    }

}

Output:



Assignment no. 4

Aim:- Write a Program in Solidity to create students data.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.6 ;

contract Student\_management{

    struct Student{

        int stud\_id;

        string Name;

        string Department;

    }

    Student[] Students;

    function add\_stud(int stud\_id, string memory Name, string memory Department) public{

        Student memory stud = Student(stud\_id, Name, Department);

        Students.push(stud);

    }

    function getStudent(int stud\_id) public view returns(string memory, string memory){

        for(uint i = 0; i < Students.length; i++){

            Student memory stud = Students[i];

            if(stud.stud\_id == stud\_id){

                return(stud.Name, stud.Department);

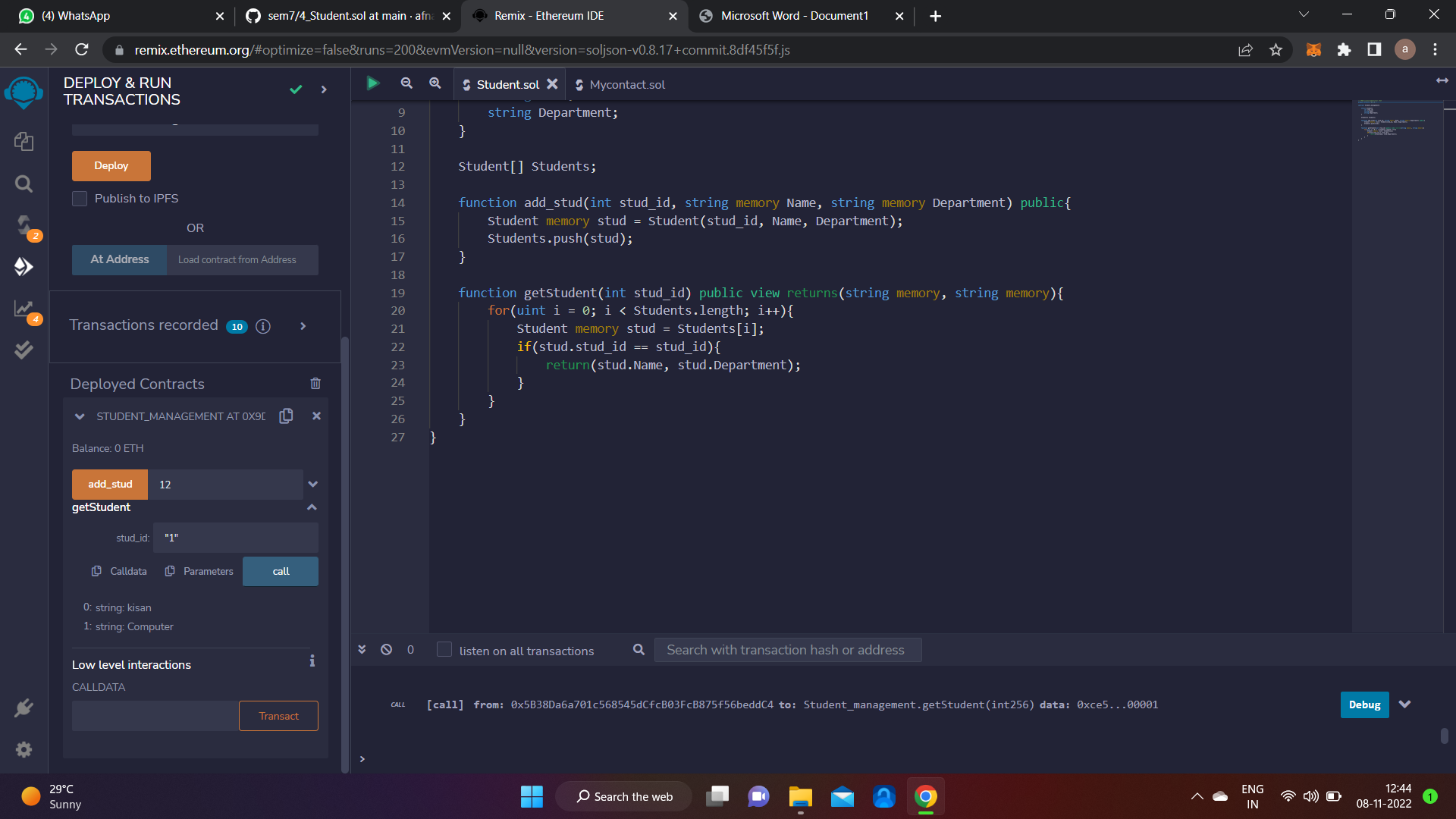
            }

        }

    }

}

Output:



Mini Project: Create a d-App for e-Voting System

Code :

pragma solidity ^0.7.6 ; // This defines the solidity version in which the code

// SPDX-License-Identifier: MIT

contract Ballot { // contract named ballot

 // VARIBLES

 struct vote { // struct = dictionary in solidity

 address voterAddresss; // used address datatype to store voter address

 bool choice; // bool value to give the choice of voter

 }

 struct voter {

 string voterName; // string dt used to store name of the voter

 bool voted; // to check wether the voter voted or not

 }

 // some variables to store some values

 uint private countResult = 0; // private var to count the result

 uint public finalResult = 0; // public var to display the result

 uint public totalVoter = 0; // public var to display the total no of voters

 uint public totalVote = 0; // public var to display no of total votes

 address public ballotOfficialAddress; // used address datatype to store the sys

 string public ballotOfficalName; // str used to store name of the ballot

 string public proposal; // proposal for which voting is done

 mapping(uint => vote) private votes; // mapping integer to vote (like in a

 mapping(address => voter) public voterRegister; // mapping address to the

 enum State { Created, Voting, Ended }

 // enums keep track of state (data types that restrict the variable to have only

 State public state;

 // MODIFIER

 modifier condition(bool \_condition) { //modifiers = conditional statements in

 \_;

 }

 modifier onlyOfficial() {

 require(msg.sender == ballotOfficialAddress);

 // condition set to ensure only official/admin address can send message

 \_;

 }

 modifier inState(State \_state) {

 require(state == \_state);

 // condition set to ensure if the global state is matching the passed state

 \_;

 }

 // FUNCTION

 constructor(string memory \_ballotofficalName,string memory \_proposal) {

 // created constructor that takes above arguments, underscores are used in

 // this const enables the admin to register himself as the official using his

 ballotOfficialAddress = msg.sender;

 ballotOfficalName = \_ballotofficalName;

 proposal = \_proposal; // used to create proposal

 state = State.Created; // used to change state

 }

 //function created to add voter takes voter address and name as arguments

 function addVoter(address \_voterAdress, string memory \_voterName )

 public

 inState(State.Created) // checks whether if someone is registered as an

 onlyOfficial // specifies that only official can register new voters

 {

 voter memory v; // calling struct to create voter record

 v.voterName = \_voterName;

 v.voted = false; // to specift that new voter hasnt voted

 voterRegister[\_voterAdress] = v;

 totalVoter++; // to move index of the struct for new record

 }

 function startVote() // function created to start voting

 public

 inState(State.Created) // the state must be in created state to move to voting

 onlyOfficial // only official can change state

 {

 state = State.Voting; // changes state from creating to voting state

 }

 function doVote(bool \_choice) // function created to enable people to vote

 public

 inState(State.Voting) // the current function can only be called in voting state

 returns (bool voted) // returns if the voter has votesd or not

 {

 bool isFound = false; // voter can only vote if the his voted state = false

 if(bytes(voterRegister[msg.sender].voterName).length != 0

 && voterRegister[msg.sender].voted == false )

 // to check if the voter has name and has not voted before

 {

 voterRegister[msg.sender].voted = true;

 vote memory v; // to store the voters vote in the struct

 v.voterAddresss = msg.sender;

 v.choice = \_choice;

 if(\_choice) {

 countResult++; // increment the result counter

 }

 votes[totalVote] = v; //to calculate total number of people who voted

 totalVote++; // increment vote if voted = true

 isFound = true;

 }

 return isFound;

 }

 function endVote() // function created to end voting

 public

 inState(State.Voting) // the current function can only be called in voting state

 onlyOfficial // only official can change state

 {

 state = State.Ended; // state changed to ended so voting ends

 finalResult = countResult; // return the result after voting

 }

}

Output: