**PRACTICAL: 6**

**Aim: Create a Smart Contract for a banking application in solidity which allows users to do the following: Mint money into your account Withdraw money from your account Send money from your account to smart contact address Check balance After a contract is created, deploy the contract on Ethereum Testnet network.**

**Code:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

//special entity called owner, make it public so every user know whos the owner

contract Banking {

address public owner;

//Mapping user address to check there balance, its public means anyone can see the ledger

mapping(address => uint256) public balances;

// Event broadcast the (User, Minted coins)

event Mint(address indexed user, uint256 amount);

//Modifier onlyOwner can modify, others are rejected

modifier onlyOwner() {

require(msg.sender == owner, "Your are not the owner");

\_;

}

// Constructor, user who cvreated the vault is the owner

constructor() {

owner = msg.sender;

}

// Function to mint new tokens & deposite to any user

function mint(address \_to, uint256 \_amount) public onlyOwner {

require(\_to != address(0), "Invalid address");

require(\_amount > 0, "Amount should be graeter than zero");

// Increase the balance of the balance of the specfied use address

balances[\_to] += \_amount;

// Emit the mint event to all users

emit Mint(\_to, \_amount);

}

// Function to check the balance of any user address by inquirer

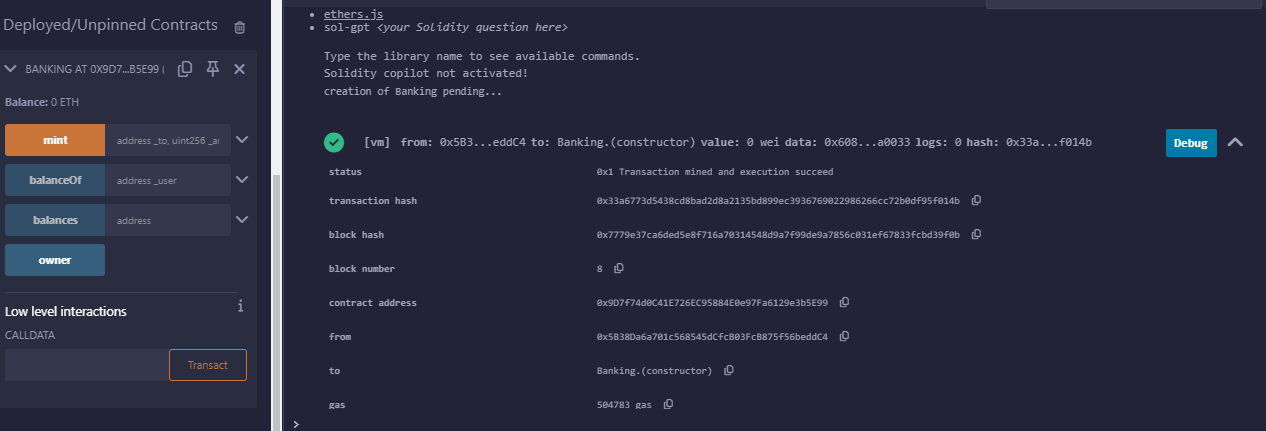
function balanceOf(address \_user) public view returns (uint256) {

return balances[\_user];

}

}

**Output:**



**PRACTICAL: 7**

**Aim: How to build a smart contract that user book rooms and pay them with cryptocurrency.**

**Code:**

//Hotel Booking Smart Contract

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// A contract is created & the Owner its Guardian is Owner.

contract HotelBooking {

    address public owner;

    uint public roomCount = 0;

//Blue print Room, which describe its Unique id, price & status

    struct Room {

        uint id;

        uint price;

        bool isBooked;

    }

//To manage rooms a magic Ledger 'mapping' links Rooms ID to its Details

    mapping(uint => Room) public rooms;

    mapping(uint => address) public bookings;

//Whenever Room Added, Booked & Received Payment, its broadcasted to all nodes

    event RoomAdded(uint id, uint price);

    event RoomBooked(uint id, address booker);

    event PaymentReceived(uint id, address from, uint amount);

//Modifier establishes rule that only owner had certain powers

    modifier onlyOwner() {

        require(msg.sender == owner, "Only owner can call this function.");

        \_;

    }

    modifier roomExists(uint \_roomId) {

        require(\_roomId > 0 && \_roomId <= roomCount, "Room does not exist.");

        \_;

    }

//This constructor declares owner as the real guardian of Hotel

    constructor() {

        owner = msg.sender;

    }

//Owner can add new rooms and emit to all nodes

    function addRoom(uint \_price) public onlyOwner {

        roomCount++;

        rooms[roomCount] = Room(roomCount, \_price, false);

        emit RoomAdded(roomCount, \_price);

    }

//Other nodes can book rooms by calling this function

    function bookRoom(uint \_roomId) public payable roomExists(\_roomId) {

           Room storage room = rooms[\_roomId];

           require(!room.isBooked, "Room is already booked");

           require(msg.value >= room.price, "Not enough Ether provided.");

           room.isBooked = true;

           bookings[\_roomId] = msg.sender;

           emit RoomBooked(\_roomId, msg.sender);

           emit PaymentReceived(\_roomId, msg.sender, msg.value);

    }

//Any node can check the booking status by calling this function

    function checkBookingStatus(uint \_roomId) public view roomExists(\_roomId)

returns (bool) {

        return rooms[\_roomId].isBooked;

    }

//Owner can withdraw all the Ether by calling this function

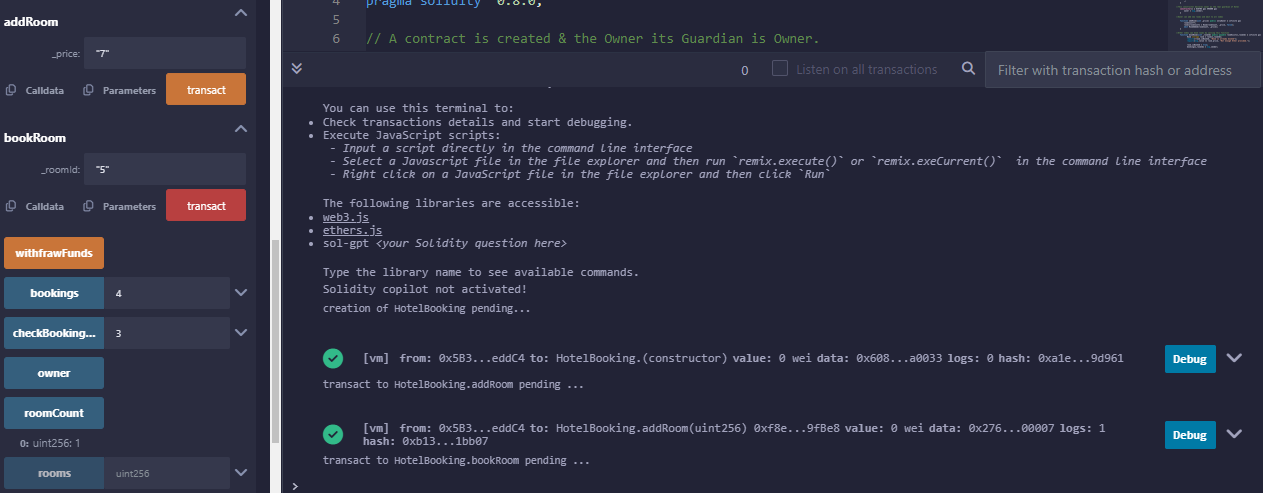
    function withfrawFunds() public onlyOwner {

        payable(owner).transfer(address(this).balance);

    }

}

**Output:**

****

**PRACTICAL: 8**

**Aim: Deploy a smart contract using MyEtherWallet (MEW).**

**Code:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract FlightReservation {

    //struct to represent a flight

    struct Flight {

        string flightNumber; // Flight number (unique identifier for the flight)

        string origin; // Origin of the flight

        string destination; // Destination of the flight

        uint256 departureTime; // Departure time (timestamp)

        uint256 seatsAvailable; //Number of available seats

    }

    // Struct to represent a booking

    struct Booking {

        address passenger; // Address of the passenger

        uint256 flightIndex; // Index of the flight in the flights array

        bool isBooked; // Indicates if the booking is active

    }

    // Array to store all the flights

    Flight[] public flights;

    // Mapping to store bookings with a unique booking ID

    mapping(uint256 => Booking) public bookings;

    // Counter for booking IDs

    uint256 public bookingIdCounter;

    // Event to notify when a flight is added

    event FlightAdded(string flightNumber, string origin, string destination,

uint256 departureTime, uint256 seatsAvailable);

    // Event to notify when a flight is booked

    event FlightBooked(uint256 bookingId, address passenger, string

flightNumber);

    // Event to notify when a booking is canceled

    event BookingCanceled(uint256 bookingId, address passenger, string

flightNumber);

    // Function to add a new flight (only accessible by the contract owner)

    function addFlight(string memory \_flightNumber, string memory \_origin, string

memory \_destination, uint256 \_departureTime, uint256 \_seatsAvailable) public {

        Flight memory newFlight = Flight({

            flightNumber: \_flightNumber,

            origin: \_origin,

            destination: \_destination,

            departureTime: \_departureTime,

            seatsAvailable: \_seatsAvailable

        });

        flights.push(newFlight);

        emit FlightAdded(\_flightNumber, \_origin, \_destination, \_departureTime,

\_seatsAvailable);

}

    // Function to book a flight

    function bookFlight(uint256 \_flightIndex) public {

        require(\_flightIndex < flights.length, "Flight does not exist");

        Flight storage flight = flights[\_flightIndex];

        require(flight.seatsAvailable > 0, "No seats available");

        // Create a new booking

        bookingIdCounter++;

        bookings[bookingIdCounter] = Booking({

            passenger: msg.sender,

            flightIndex: \_flightIndex,

            isBooked: true

        });

        // Decrease the number of available seats

        flight.seatsAvailable--;

        emit FlightBooked(bookingIdCounter, msg.sender, flight.flightNumber);

    }

    // Function to cancel a booking

    function cancelBooking(uint256 \_bookingId) public {

        require(bookings[\_bookingId].isBooked, "Booking does not exist or is  already canceled");

        require(bookings[\_bookingId].passenger == msg.sender, "Only the passenger can cancel their booking");

        // Retrieve the flight index from the booking

        uint256 flightIndex = bookings[\_bookingId].flightIndex;

        // Mark the booking as canceled

        bookings[\_bookingId].isBooked = false;

        // Increase the number of available seats for the flight

        flights[flightIndex].seatsAvailable++;

        emit BookingCanceled(\_bookingId, msg.sender,

flights[flightIndex].flightNumber);

    }

    // Function to get the details of a flight

    function getFlightDetails(uint256 \_flightIndex) public view returns (string

memory flightNumber, string memory origin, string memory destination, uint256

departureTime, uint256 seatsAvailable) {

        require(\_flightIndex < flights.length, "Flight does not exist");

        Flight storage flight = flights[\_flightIndex];

        return (flight.flightNumber, flight.origin, flight.destination,

flight.departureTime, flight.seatsAvailable);

    }

    // Function to get the details of a booking

    function getBookingDetails(uint256 \_bookingId) public view returns (address

passenger, string memory flightNumber, bool isBooked) {

        require(bookings[\_bookingId].isBooked, "Booking does not exist or is canceled");

        Booking storage booking = bookings[\_bookingId];

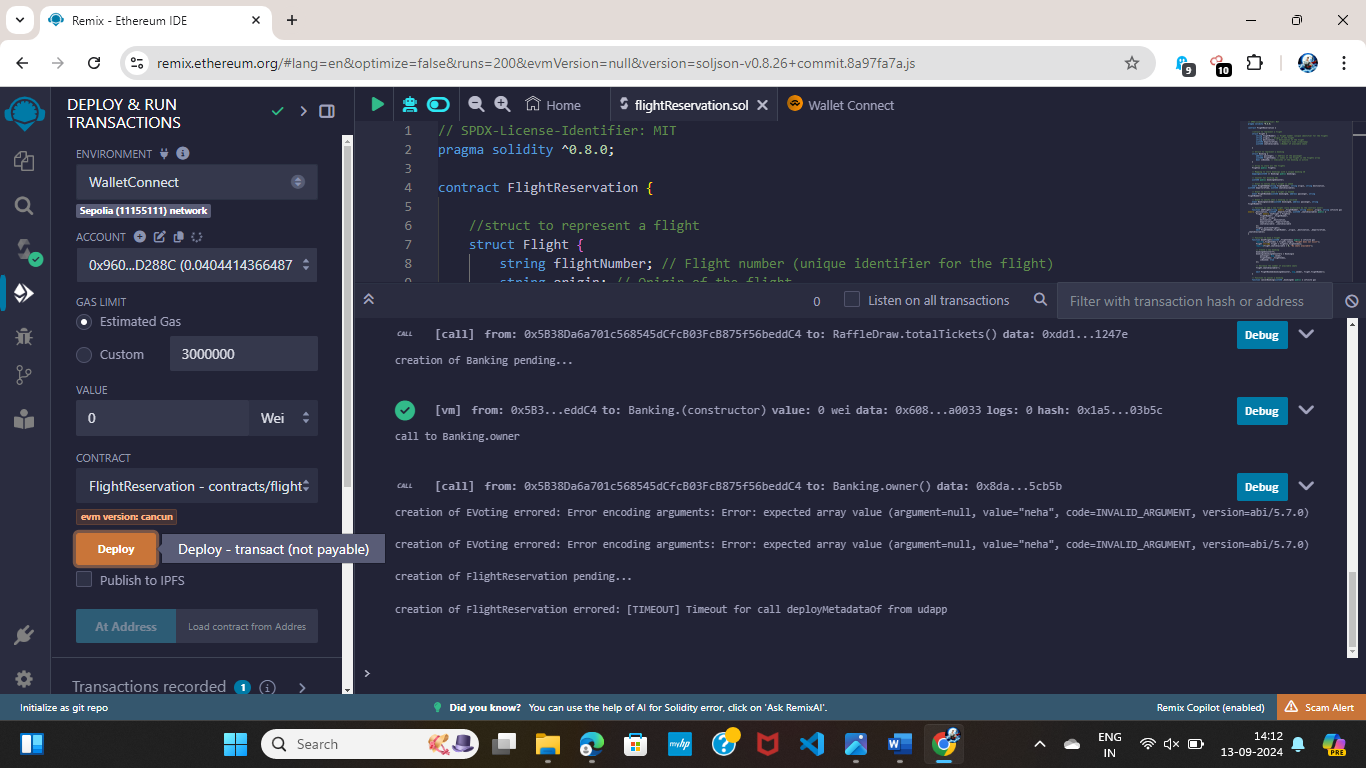
        Flight storage flight = flights[booking.flightIndex];

        return (booking.passenger, flight.flightNumber, booking.isBooked);

    }

}

**Output:**

****

**PRACTICAL: 9**

**Aim: Deploy the smart contract for RAFFLE DRAW GAME.**

**Code:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract RaffleDraw {

address public owner;

address[] public participants;

uint256 public ticketPrice;

uint256 public totalTickets;

uint256 public ticketsSold;

bool public raffleEnded;

address public winner;

event TicketPurchased(address indexed participant, uint256 indexed ticketNumber);

event WinnerSelected(address indexed winner);

event RaffleEnded();

constructor(uint256 \_ticketPrice, uint256 \_totalTickets){

owner = msg.sender;

ticketPrice = \_ticketPrice;

totalTickets = \_totalTickets;

raffleEnded = false;

}

modifier onlyOwner(){

require(msg.sender == owner, "only owner can call this function ");

\_;

}

modifier raffleActive(){

require(!raffleEnded, "Raffle has already ended");

\_;

}

function purchaseTicket() public payable raffleActive{

require(msg.value == ticketPrice, "Incorrect ticket price");

require(ticketsSold < totalTickets, "All tickets have been sold");

participants .push(msg.sender);

ticketsSold += 1;

emit TicketPurchased(msg.sender, ticketsSold);

}

function endRaffle() public onlyOwner raffleActive{

require(ticketsSold == totalTickets,"Raffle can't end untill all Tickets are sold");

raffleEnded =true;

emit RaffleEnded();

}

function selectWinner() internal {

require(raffleEnded,"Raffle has not ended yet");

uint256 randomIndex = uint256(keccak256(abi.encodePacked(block.timestamp,block.difficulty,participants))) % participants.length;

winner = participants[randomIndex];

emit WinnerSelected(winner);

payable (winner).transfer(address(this).balance);

}

function getParticipants() public view returns (address[]memory){

return participants;

}

function grtWinner() public view returns (address){

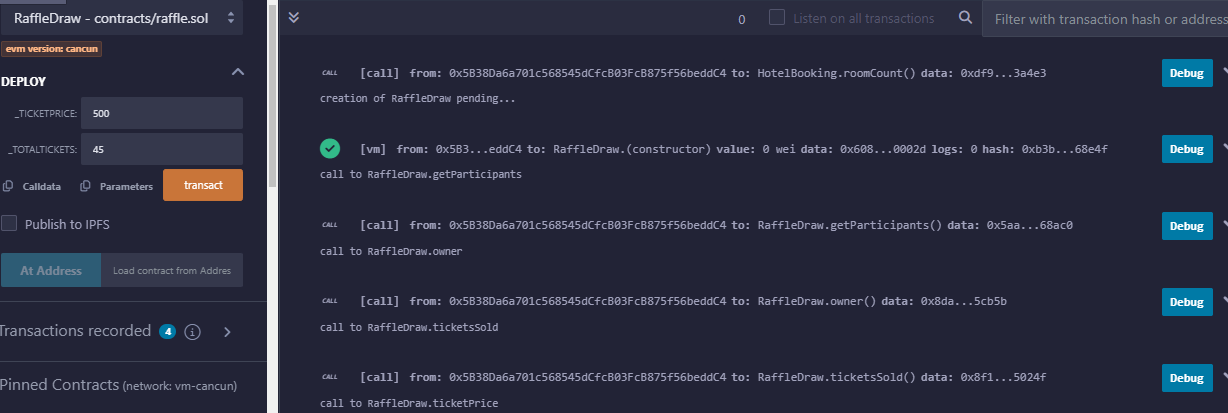
require(raffleEnded,"Raffle has not ended yet");

return winner;

}

}

**Output:**



**PRACTICAL: 10**

**Aim: Deploy Smart Contract for E-Voting**

**Code:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract EVoting {

     struct Voter {

        bool voted;

        uint vote;

        bool rightToVote;

        address delegate;

}

struct Proposal {

    string name;

    uint voteCount;

}

address public electionOfficial;

mapping(address => Voter) public voters;

Proposal[] public proposals;

constructor(string[] memory proposalNames) {

    electionOfficial = msg.sender;

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

        proposals.push(Proposal({

            name: proposalNames[i],

            voteCount: 0

            }));

    }

}

function giveRightToVote(address voter) public {

    require(

        msg.sender == electionOfficial,

        "Only election official can give right to vote."

        );

        require(!voters[voter].voted,

        "The voter already voted."

        );

        require(!voters[voter].rightToVote,

        "Voter already has the right to vote."

        );

        voters[voter].rightToVote = true;

}

function delegate(address to) public {

    Voter storage sender = voters[msg.sender];

    require(sender.rightToVote,"You have no right to vote");

    require(!sender.voted, "You already voted.");

    require(to != msg.sender, "Self-delegation is disallowed.");

    while (voters[to].delegate != address(0)) {

        to = voters[to].delegate;

     }

     sender.voted = true;

     sender.delegate = to;

     Voter storage delegate\_ = voters[to];

     if (delegate\_.voted) {

        proposals[delegate\_.vote].voteCount += 1;

        } else {

            delegate\_.rightToVote = true;

            }

        }

        function vote(uint proposal) public {

            Voter storage sender = voters[msg.sender];

            require(sender.rightToVote, "Has no right to vote");

            require(!sender.voted, "Already voted.");

            sender.voted = true;

            sender.vote = proposal;

            proposals[proposal].voteCount += 1;

        }

        function winningProposal() public view returns (uint winningProposal\_) {

            uint winningVoteCount = 0;

            for (uint p = 0; p < proposals.length; p++) {

                if (proposals[p].voteCount > winningVoteCount) {

                    winningVoteCount = proposals[p].voteCount;

                    winningProposal\_ = p;

                    }

                }

            }

            function winnerName() public view returns (string memory winnerName\_) {

                winnerName\_ = proposals[winningProposal()].name;

            }

}

**Output:**

