4 September 2023

CS61065 Theory and Applications of Blockchain

Assignment 1: Ethereum Basics

Date of Submission: September 15, 2023 EOD

You need to submit the assignment as a group of maximum 4 members. Only one member of each group should submit the assignment.

In this assignment, you will get familiar with the basics of Ethereum. You will learn how to connect to the Ethereum main network, access test networks, manage your accounts, and execute transactions. You will also get started with Ethereum JavaScript API (web3.js) and do some basic operations in the test networks.

Submission Instructions

Submit the answers to the questions of Part A, Part B and Part C in the following Google Forms link:

https://forms.gle/ejNJEPu11QvmnAtv7

Submit the assignment in a **group of 4 (Max)**. In case of multiple submissions, the final one before the deadline will be considered.

Mark Distribution:

Part A: 25 Marks
Part B: 5 Marks (Q1 - 1 Mark, Q2 - 2 Marks, Q3 - 2 Marks)
Part C: 30 Marks (The code should run in the demonstration without any error and marks for each individual component is mentioned in Part C Question Description). Marks will be deducted if the smart contract address is not submitted in form within the deadline.

Part A

We recommend you install **Geth** by following the instructions on the documentation page and try it out: https://geth.ethereum.org/docs/install-and-build/installing-geth

For the rest of the assignment, create an account at https://infura.io/, and use it to connect to the **Sepolia** test network.

Infura provides you with JSON-RPC over the HTTP interface. Use JSON-RPC calls to answer the following questions in the Google form link provided above. Provide (a) the answer, (b) the JSON payload used in the JSON-RPC, and (c) the response from the RPC call for each question. [Note: Each question will need a single RPC call only].

A.1. Query the current gas price in wei. Give the answer as an integer (not in hex).
Sample Answer:

I. Answer: `15877637594`

II. JSON RPC payload:
`{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":1}`

III. Response: `{"jsonrpc":"2.0","id":1,"result":"0x3b26185da"}`

A.2. Query the current latest block number (converted to decimal).

I. Answer, II. JSON RPC payload, III. Response

A3. Find the balance (In Integer) of the account of a given address.
0xBaF6dC2E647aeb6F510f9e318856A1BCd66C5e19

I. Answer, II. JSON RPC payload, III. Response

A.4. Query the information about a transaction requested by transaction hash "0xdcae4a84a5780f62f18a9afb07b3a7627b9a28aa128a76bfddec72de9a0c2606".
Find Out:
I. the number of transactions made by the sender prior to this one in the block. Give the answer as an integer.
II. value transferred in Wei(In Integer)
III. JSON RPC payload
IV. Response

A.5. Find the number of peers connected currently to your Geth client (in Infura).

I. Answer (Integer), II. JSON RPC payload, III. Response

A6. Query transaction receipt for the transaction with hash "0x5d692282381c75786e5f700c297def496e8e54f0a96d5a4447035f75085933cb".
Find out

I. the blockNumber (integer),
II. blockHash,
III. cumulativeGasUsed (integer),
IV. transactionIndex (integer).

A7. Find out the number of transactions in the block with the given block: 0x1132aea

I. Answer (Integer), II. JSON RPC payload, III. Response

Part B

Use the **Sepolia faucet [(https://sepoliafaucet.com/)](https://sepoliafaucet.com/)** to obtain some Ethereum in your account.

Use **web3.js** to query and execute the following smart contract:

Smart Contract Address: **0xF98bFe8bf2FfFAa32652fF8823Bba6714c79eDd4**

The smart contract stores your roll number corresponding to your Ethereum account address. Generate the ABI from the contract code. The code of the contract is as follows:

```
contract AddressRollMap {
    mapping(address => string) public roll;
    function update(string calldata newRoll) public
    {
        roll[msg.sender] = newRoll;
    }
    function get(address addr) public view returns (string
memory)
    {
        return roll[addr];
    }
    function getmine() public view returns (string memory)
    {
        return roll[msg.sender];
    }
}
```

B.1. Query the roll for the address: 0x328Ff6652cc4E79f69B165fC570e3A0F468fc903

B.2. Input your own roll through the **update** function.

I. **Submit your Ethereum account address from which you made the transaction,**
II. **The transaction id.**

Part C: Movie Ticket Booking with Ethereum Smart Contract

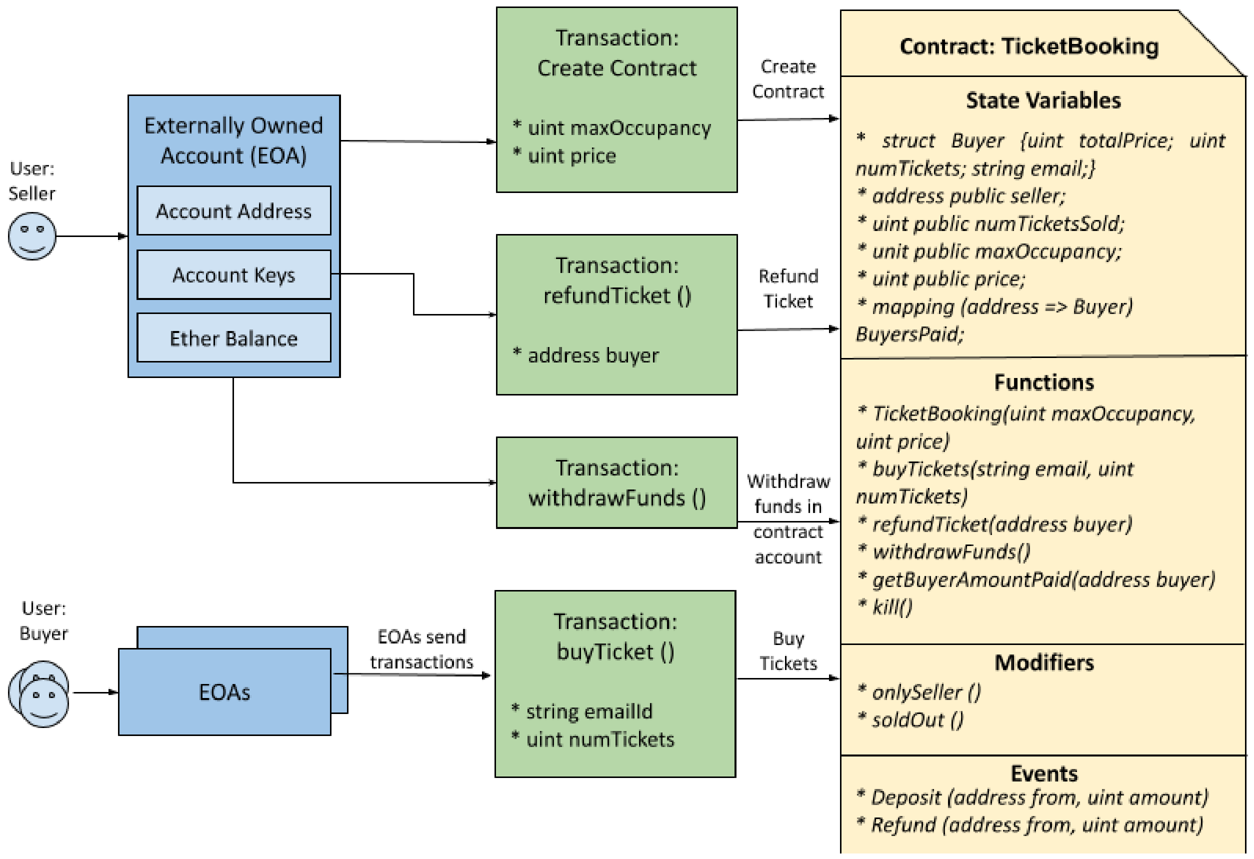
Only one member from each group should submit the assignment in Moodle. Clearly mention your group details in the submission.

In this assignment, you will get familiar with Ethereum smart contracts in solidity language. You will learn how you can use blockchain to enforce certain rules in a distributed setting in order to construct applications where the users do not necessarily trust each other.

Deploy your smart contract in the Sepolia test network. You may use the combination of Remix IDE and MetaMask extension to develop, test, and deploy your contract. Write the address of the contract and your roll numbers of all group members as comments in the solidity file. Submit the smart contract address (one for each group) in the given google form. And submit your solidity code as a single file with .sol extension as <RollNo>.sol in Moodle.

Description:

We want to develop a smart contract for Movie Ticket Booking. Using this contract the movie theater owner (seller) can sell tickets for a movie. People can purchase the movie tickets by sending transactions to the contract. You have to write an ***"TicketBooking"*** contract.


User: Seller
Externally Owned Account (EOA)
Account Address
Account Keys
Ether Balance
Transaction: Create Contract
* uint maxOccupancy
* uint price
Create Contract
Transaction: refundTicket ()
* address buyer
Refund Ticket
Transaction: withdrawFunds ()
Withdraw funds in contract account
User: Buyer
EOAs
EOAs send transactions
Transaction: buyTicket ()
* string emailId
* uint numTickets
Buy Tickets
Contract: TicketBooking
State Variables
* struct Buyer {uint totalPrice; uint numTickets; string email;}
* address public seller;
* uint public numTicketsSold;
* unit public maxOccupancy;
* uint public price;
* mapping (address => Buyer) BuyersPaid;
Functions
* TicketBooking(uint maxOccupancy, uint price)
* buyTickets(string email, uint numTickets)
* refundTicket(address buyer)
* withdrawFunds()
* getBuyerAmountPaid(address buyer)
* kill()
Modifiers
* onlySeller ()
* soldOut ()
Events
* Deposit (address from, uint amount)
* Refund (address from, uint amount)

While deploying the smart contract, the owner of the contract specifies 2 parameters - the maxOccupancy (total number of tickets to sell) and the price of each ticket(in wei).

Write a constructor as follows: **(5 Marks)**

```
constructor(uint _quota, uint _price) {
       owner = msg.sender;
       numTicketsSold = 0;
       quota = _quota;
       price = _price;
   }
```

In the smart contract, define a custom data type to represent a ***Buyer***. The **Buyer** structure maintains information related to person who book for the movie including the amount paid, the number of tickets purchased and the email address. **(3 Marks)**

The ***BuyersPaid*** variables maintains a mapping from the address of the buyer to the ***Buyer*** structure. **(2 Marks)**

To buy tickets for the movie, a user can send transaction to the ***buyTicket*** function of the contract along with the user's email address and number of tickets to buy. The user also sends a value equal to the total cost of the tickets along with the transaction. **(10 Marks)**
The function ***buyTickets*** is defined as follows:

```
function buyTicket(string memory email, uint numTickets) public
payable soldOut {
. . .
}
```

- It has a function modifier ***"soldOut"***, which check if all the movie tickets are sold out.

```
modifier soldOut() {
       require(numTicketsSold < quota, "All tickets have been
sold");
       _;
   }
```

- In the function, checks if the buyer is already bought a ticket for the movie.
  - If already bought, update the number of tickets purchased by the user and the total amount paid.
  - If not bought, add the user's address to the "***buyersPaid***" list and store the user's email address, the amount paid and the number of tickets purchased.
- If the value sent by the user is more than the total cost of the tickets, the balance should be refunded.

The amount paid by all the users for the tickets purchased is held in the contract account. The contract Seller (or benificiery) can withdraw the amount from the contract account by sending a transaction to the "***withdrawFunds***" functions. This function has a modifier ***"onlyOwner"*** to check if the transaction is sent by the owner. **(5 Marks)**

```
modifier onlyOwner() {
 . .
}
```

```
function withdrawFunds() public onlyOwner {
. . .
}
function refundTicket(address buyer) public onlyOwner {
. . .
}
```

Additional Functions:

In addition to the function, also write the following functions to debug and test the smart contract:

```
function getBuyerAmountPaid(address buyer) public view returns (uint)
```

Return the total amount paid by a buyer.

```
function kill() public onlyOwner
```

To kill the movie ticket booking event.

References

Solidity documentation: https://docs.soliditylang.org/en/v0.8.7/

Remix Editor: https://remix.ethereum.org/