Lab 1/Project 1 – Introduction to Smart Contract Development

COMP6452 Software Architecture for Blockchain Applications

2023 Term 2

1 Learning Outcomes

In this lab, which also leads to Project 1, you will learn how to write a smart contract using Solidity and deploy it on the Ethereum testnet blockchain. After completing the lab/project, you will be able to:

- develop a simple smart contract using Solidity
- test your smart contract manually and automatically by issuing transactions and running unit tests
- create and fund your account on the Ethereum testnet
- deploy your contract to the Ethereum testnet

This lab provides step-by-step instructions to develop, test, and deploy your first smart contract. Then as Project 1, *which is graded*, you will extend that smart contract to fix several defects and add additional functionality.

2 Introduction

Smart contracts are user-defined code deployed on and executed by nodes in a blockchain. In addition to executing instructions, smart contracts can hold, manage, and transfer digitalised assets. For example, a smart contract could be seen as a bunch of if/then conditions that are an algorithmic implementation of a financial service such as trading, lending, and insurance.

Smart contracts are deployed to a blockchain as transaction data. Execution of a smart contract function is triggered using a transaction issued by a user (or a system acting on behalf of a user) or another smart contract, which was in turn triggered by a user-issued transaction. A smart contract does not auto-execute and must be triggered using a user-issued transaction. Inputs to a smart contract function are provided through a transaction and the current state of the blockchain. Due to blockchains' immutability, transparency, consistency, and integrity properties, smart contract code is immutable and deterministic making its execution trustworthy. While "code is law" [1] is synonymous with smart contracts, smart contracts are neither smart nor legally binding per the contract law. However, they can be used to execute parts of a legal contract.

While Bitcoin [2] supports an elementary form of smart contracts, it was Ethereum [3] that demonstrated the true power of smart contracts by developing a Turing complete language and a run-time environment to code and execute smart contracts. Smart contracts in Ethereum are deployed and executed as *bytecode*, i.e., binary code results from compiling code written in a high-level language. Bytecode runs on each blockchain node's Ethereum Virtual Machine (EVM) [4]. This is analogous to Java bytecode executing on Java Virtual Machine (JVM).

Solidity [5] is the most popular smart contract language for Ethereum. Contracts in Solidity are like classes in object-oriented languages, and contracts deployed onto the blockchain are like objects. A Solidity smart contract contains persistent data in state variables, and functions that can access and modify these state variables. A deployed contract resides at a specific address on the Ethereum blockchain. Solidity is a high-level, object-oriented language that is syntactically similar to JavaScript. It is statically typed and supports inheritance, libraries, and user-defined types. As Solidity code is ultimately compiled into Ethereum bytecode, other blockchain platforms that support the EVM, such as Hyperledger Besu, can also execute it.

Fig. 1 shows the typical development cycle of a smart contract. Like any program, it starts with requirement analysis and modelling. State diagrams, Unified Modelling Language (UML), and Business

Process Model and Notation (BPMN) are typically used to model smart contracts. The smart contract code is then developed using a suitable tool ranging from Notepad to sophisticated IDEs. Various libraries and Software Development Kits (SDKs) may be used to minimise errors and enhance productivity. Depending on the smart contract language, code may also need to be compiled, e.g., Solidity. Smart contract code and results must be bug-free because they are immutable and transparent.

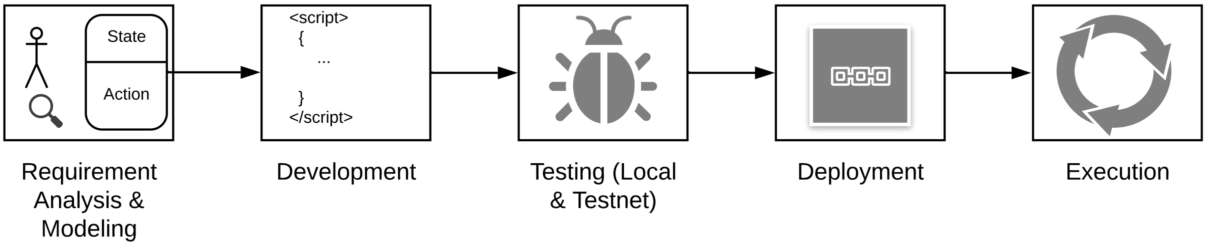

Figure 1: Smart contract development cycle.

Because transactions trigger smart contract functions, we need to pay fees to deploy and execute smart contracts on a public blockchain. In Ethereum, this fee is referred to as gas. More formally, *gas* is a unit of account within the EVM used in calculating a transaction's fee, which is the amount of Ether (ETH) a transaction's sender must pay to the miner who includes the transaction in the blockchain. The amount of gas needs to execute a smart contract depends on several factors such as the computational complexity of the code, the volume of data in memory and storage, and bandwidth requirements. There is also a cost to deploy a smart contract depending on the length of the bytecode. Therefore, it is essential to extensively test and optimise a smart contract to keep the cost low. The extent that one can test (e.g., unit testing), debug, and optimise the code depends on the smart contract language and tool availability. While Ethereum has a rich set of tools, in this lab, we will explore only a tiny subset of them.

Most public blockchains also host a test/development network, referred to as the *testnet*, that is identical in functionality to the production network. Further, they usually provide fast transaction finality and do not charge real transaction fees. It is highly recommended to test a smart contract on a testnet. Testnet can also be used to estimate transaction fees you may need to pay in the production network.

Once you are confident that the code is ready to go to the production/public blockchain network, the next step is to deploy the code using a transaction. Once the code is successfully deployed, you will get an address (aka identifier or handler) for future interactions with the smart contract. Finally, you can interact with the smart contract by issuing transactions with the smart contract address as the recipient (i.e., `to` address). The smart contract will remain active until it is disabled, or it reaches a terminating state. Due to the immutability of blockchains, smart contract code will remain in the blockchain even though it is deactivated and cannot be executed.

This lab has two parts. In part one (Section 3 to 8), you will develop, test, and deploy a given smart contract to the Ethereum Sepolia testnet by following step-by-step instructions. Part two (Section 9) is Project 1 where you will update the smart contract and unit tests to fix some of its functional weaknesses, and then deploy it onto the testnet. You are not expected to complete this handout during tutorial/lab time. Instead, you will need additional time alone to complete the lab and Project 1. Tutors will provide online and offline support.

3 Developing a Smart Contract

In this lab, we will write a smart contract and deploy it to the public Ethereum Sepolia testnet. The motivation of our Decentralised Application (DApp) is to solve a million-Dollar question: *Where to have lunch?*

The basic requirements for our DApp to determine the lunch venue are as follows:

1. The contract deployer SHOULD be able to nominate a list of restaurants `r` to vote for.

2. The contract deployer SHOULD be able to create a list of voters/friends v who can vote for `r` restaurants.

3. A voter MUST be able to cast a vote only once.

4. The contract MUST stop accepting votes when the quorum is met (e.g., $number\_of\_votes > |v|/2$) and declare the winning restaurant as the lunch venue.

The following code shows a smart contract written in Solidity to decide the lunch venue based on votes. Line 1 is a machine-readable license statement that indicates that the source code is unlicensed. Lines starting with `//`, `///`, and `/**` are comments.

```
1  /// SPDX-License-Identifier: UNLICENSED
2
3  pragma solidity ^0.8.0;
4
5  /// @title Contract to agree on the lunch venue
6  /// @author Dilum Bandara, CSIRO's Data61
7
8  contract LunchVenue{
9
10     struct Friend {
11         string name;
12         bool voted; //Vote state
13     }
14
15     struct Vote {
16         address voterAddress;
17         uint restaurant;
18     }
19
20     mapping (uint => string) public restaurants; //List of restaurants (restaurant no, name)
21     mapping(address => Friend) public friends;  //List of friends (address, Friend)
22     uint public numRestaurants = 0;
23     uint public numFriends = 0;
24     uint public numVotes = 0;
25     address public manager;                     //Contract manager
26     string public votedRestaurant = "";         //Where to have lunch
27
28     mapping (uint => Vote) public votes;        //List of votes (vote no, Vote)
29     mapping (uint => uint) private _results;    //List of vote counts (restaurant no, no of votes)
30     bool public voteOpen = true;                //voting is open
31
32     /**
33      * @dev Set manager when contract starts
34      */
35     constructor () {
36         manager = msg.sender;                   //Set contract creator as manager
37     }
38
39     /**
40      * @notice Add a new restaurant
41      * @dev To simplify the code, duplication of restaurants isn't checked
42      *
43      * @param name Restaurant name
44      * @return Number of restaurants added so far
45      */
46     function addRestaurant(string memory name) public restricted returns (uint){
47         numRestaurants++;
48         restaurants[numRestaurants] = name;
49         return numRestaurants;
50     }
51
52     /**
53      * @notice Add a new friend to voter list
54      * @dev To simplify the code duplication of friends is not checked
55      *
56      * @param friendAddress Friend's account/address
57      * @param name Friend's name
58      * @return Number of friends added so far
59      */
60     function addFriend(address friendAddress, string memory name) public restricted returns (uint){
61         Friend memory f;
62         f.name = name;
63         f.voted = false;
```

```
        friends[friendAddress] = f;
        numFriends++;
        return numFriends;
    }

    /**
     * @notice Vote for a restaurant
     * @dev To simplify the code duplicate votes by a friend is not checked
     *
     * @param restaurant Restaurant number being voted
     * @return validVote Is the vote valid? A valid vote should be from a registered friend to a registered restaurant
    */
    function doVote(uint restaurant) public votingOpen returns (bool validVote){
        validVote = false;                                  //Is the vote valid?
        if (bytes(friends[msg.sender].name).length != 0) {  //Does friend exist?
            if (bytes(restaurants[restaurant]).length != 0) {   //Does restaurant exist?
                validVote = true;
                friends[msg.sender].voted = true;
                Vote memory v;
                v.voterAddress = msg.sender;
                v.restaurant = restaurant;
                numVotes++;
                votes[numVotes] = v;
            }
        }

        if (numVotes >= numFriends/2 + 1) { //Quorum is met
            finalResult();
        }
        return validVote;
    }

    /**
     * @notice Determine winner restaurant
     * @dev If top 2 restaurants have the same no of votes, result depends on vote order
    */
    function finalResult() private{
        uint highestVotes = 0;
        uint highestRestaurant = 0;

        for (uint i = 1; i <= numVotes; i++){   //For each vote
            uint voteCount = 1;
            if(_results[votes[i].restaurant] > 0) { // Already start counting
                voteCount += _results[votes[i].restaurant];
            }
            _results[votes[i].restaurant] = voteCount;

            if (voteCount > highestVotes){ // New winner
                highestVotes = voteCount;
                highestRestaurant = votes[i].restaurant;
            }
        }
        votedRestaurant = restaurants[highestRestaurant];   //Chosen restaurant
        voteOpen = false;                                   //Voting is now closed
    }

    /**
     * @notice Only the manager can do
     */
    modifier restricted() {
        require (msg.sender == manager, "Can only be executed by the manager");
        _;
    }

    /**
     * @notice Only when voting is still open
     */
    modifier votingOpen() {
        require(voteOpen == true, "Can vote only while voting is open.");
        _;
    }
}
```

Line 3 tells that the code is written for Solidity and should not be used with a compiler earlier than version 0.8.0. The ∧ symbol says that the code is not designed to work on future compiler versions, e.g., 0.9.0. It should work on any version labelled as `0.8.xx`. This ensures that the contract is not compilable with a new (breaking) compiler version, where it may behave differently. These constraints are indicated using the `pragma` keyword, an instruction for the compiler. As Solidity is a rapidly evolving language and smart contracts are immutable, it is desirable even to specify a specific version such that all contract participants clearly understand the smart contract's behaviour. You can further limit the compiler version using greater and less than signs, e.g., `pragma solidity >=0.8.2 <0.9.0`.

In line 8, we declare our smart contract as `LunchVenue`. The smart contract logic starts from this line and continues till line 135 (note the opening and closing brackets and indentation). Between lines 10 and 18, we define two structures that help to keep track of a list of friends and votes. We keep track of individual votes to avoid non-repudiation. Then we define a bunch of variables between lines 20 and 30. The `address` is a special data type in Solidity that refers to a 160-bit Ethereum address/account. An address could refer to a user or a smart contract. `string` and `bool` have usual meanings. `uint` stands for unsigned integer data type, i.e., nonnegative integers.

Lines 20-21 and 28-29 define several hash maps (aka map, hash table, or key-value store) to keep track of the list of restaurants, friends, votes, and results. A hash map is like a two-column table, e.g., in the `restaurants` hash map the first column is the restaurant number and the second column is the restaurant name. Therefore, given the restaurant number, we can find its name. Similarly, in `friends`, the first column is the Ethereum address of the friend, and the second column is the `Friend` structure that contains the friend's name and vote status. Compared to some of the other languages, Solidity cannot tell us how many keys are in a hash map or cannot iterate on a hash map. Thus, the number of entries is tracked separately (lines 22-24). Also, a hash map cannot be defined dynamically.

The `manager` is used to keep track of the smart contract deployer's address. Most voted restaurant and vote open state are stored in variables `votedRestaurant` and `voteOpen`, respectively.

Note the permissions of these variables. `_results` variable is used only when counting the votes to determine the most voted restaurant. Because it does not need to be publically accessible it is marked as a private variable. Typically, private variables start with an underscore. All other three variables are public. Public variables in a Solidity smart contract can be accessed through smart contract functions, while private variables are not. The compiler automatically generates getter functions for public variables.

Lines 35-37 define the *constructor*, a special function executed when a smart contract is first created. It can be used to initialise state variables in a contract. In line 36, we set the transaction/message sender's address (`msg.sender`) that deployed the smart contract as the contract's `manager`. The `msg` variable (together with `tx` and `block`) is a special global variable that contains properties about the blockchain. `msg.sender` is always the address where the external function call originates from.

`addRestaurant` and `addFriend` functions are used to populate the list of restaurants and friends that can vote for a restaurant. Each function also returns the number of restaurants and friends added to the contract, respectively. The `memory` keyword indicates that the `name` is a string that should be held in the memory as a temporary value. In Solidity, all string, array, mapping, and struct type variables must have a data location (see line 61). EVM provides two other areas to store data, referred to as `storage` and `stack`. For example, all the variables between lines 20 and 30 are maintained in the storage, though they are not explicitly defined.

`restricted` is a *function modifier*, which is used to create additional features or to apply restrictions on a function. For example, the `restricted` function (lines 123-126) indicates that only the `manager` can invoke this function. Therefore, function modifiers can be used to enforce access control. If the condition is satisfied, the function body is placed on the line beneath `_;`. Similarly, `votingOpen` (lines 131-134) is used to enforce that votes are accepted only when `voteOpen` is `true`.

The `doVote` function is used to vote, insofar as the voting state is open, and both the friend and restaurant are valid (lines 76-94). The voter's account is not explicitly defined, as it can be identified from the transaction sender's address (line 78). This ensures that only an authorised user (attested through their digital signature attached to the transaction) can transfer tokens from their account. It further returns a Boolean value to indicate whether voting was successful. In line 90, after each vote, we check whether the quorum is reached. If so, the `finalResults` private function is called to choose the most voted reataurant. This function uses a hash map to track the vote count for each restaurant, and the one with the highest votes is declared as the lunch venue. The voting state is also marked as no longer open by setting `voteOpen` to `false` (line 117).

Let us now create and compile this smart contract. For this, we will use Remix IDE, an online Integrated Development Environment (IDE) for developing, testing, deploying, and administering smart contracts for Ethereum-like blockchains. Due to zero setup and a simple user interface, it is a good learning platform for smart contract development.

Step 1. Using your favourite web browser, go to `https://remix.ethereum.org/`.

Step 2. Click on the `File explorer` icon (symbol with two documents) from the set of icons on the left. Select the `contracts` folder – the default location for smart contracts on Remix. Then click on `Create new file` icon (small document icon), and enter `LunchVenue.sol` as the file name.

Alternatively, you can click on the `New File` link in `Home` tab. If the file is created outside the `contracts` folder, make sure to move it into the `contracts` folder.

Step 3. Type the above smart contract code in the editor. Better not copy and paste the above code from PDF, as it may introduce hidden characters or unnecessary spaces preventing the contract from compiling.

Step 4. As seen in Fig. 2, set the compiler options are as follows, which can be found under `Solidity compiler` menu option on the left:

- Compiler – `0.8.5+....` (any commit option should work)
- In `Advanced Configurations` select `Compiler configuration`
  - Language – `Solidity`
  - EVM Version – `default`
- Make sure `Hide warnings` is not ticked. Others are optional.

Step 5. Then click on the `Compile LunchVenue.sol` button. Carefully fix any errors and warnings. While Remix stores your code on the browser storage, it is a good idea to link it to your Github account. You may also save a local copy.

Step 6. Once compiled, you can access the binary code by clicking on the `Bytecode` link at the bottom left, which will copy it to the clipboard. Paste it to any text editor to see the binary code and EVM instructions (opcode).

Similarly, using the `ABI` link, you can check the Application Binary Interface (ABI) of the smart contract. ABI is the interface specification to interact with a contract in the Ethereum ecosystem. Data are encoded as a JSON (JavaScript Object Notation) schema that describes the set of functions, their parameters, and data formats. Also, click on the `Compilation Details` button to see more details about the contract and its functions.

4 Deploying the Smart Contract

First, let us test our smart contract on Remix JavaScript VM to ensure that it can be deployed without much of a problem. Remix JavaScript VM is a simulated blockchain environment that exists in your browser. It also gives you 10 pre-funded accounts to test contracts. Such simulated testing helps us validate a smart contract's functionality and gives us an idea about the transaction fees.

Ethereum defines the transaction fee as follows:

$$transaction\,fee = gas\,limit \times gas\,price \tag{1}$$

The *gas limit* defines the maximum amount of gas we are willing to pay to deploy or execute a smart contract. This should be determined based on the computational and memory complexity of the code, the volume of data it handles, and bandwidth requirements. If the gas limit is set too low, the smart contract could terminate abruptly as it runs out of gas. If it is too high, errors such as infinite loops could consume all our Ether. Hence, it is a good practice to estimate the `gas limit` and set a bit higher value to accommodate any changes during the execution (it is difficult to estimate the exact gas limit as the execution cost depends on the state of the blockchain).

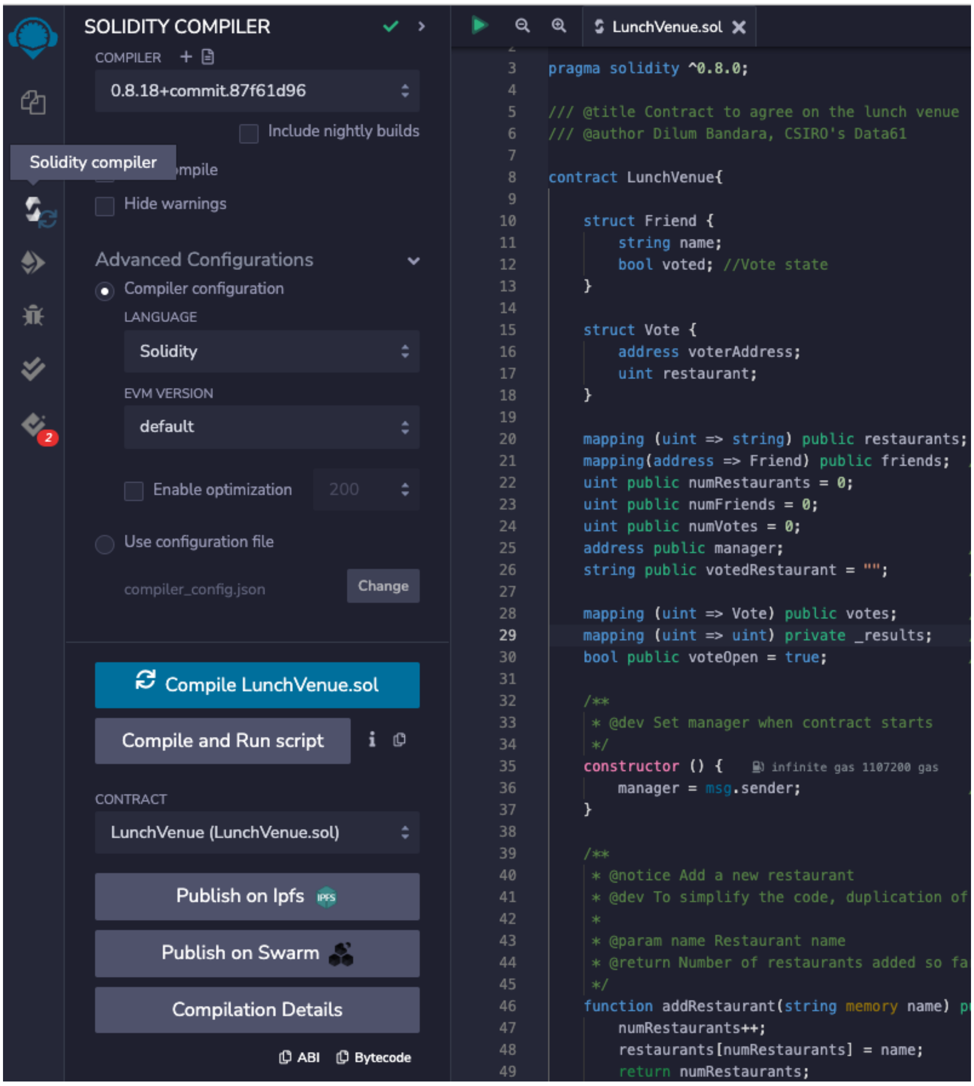

Figure 2: Compiler options.

The *gas price* determines how much we are willing to pay for a unit of gas. When a relatively higher gas price is offered, the time taken to include the transaction in a block typically reduces. Most blockchain explorers, such as Etherscan.io, provide statistics on market demand for gas price. It is essential to consider such statistics when using the Ethereum production network to achieve a good balance between transaction latency and cost.

Step 7. Select `Deploy & run transactions` menu option on the left. Then set the options as follows (see Fig. 3):

- Environment – `Remix VM (Shanghai)`
- Account – Pick one of the accounts with some Ether
- Gas Limit – `3000000` (use the default)
- Value – `0` (we are not transferring any Ether to the smart contract)
- Contract – `LunchVenue`

Step 8. Click on the `Deploy` button. This should generate a transaction to deploy the `LunchVenue` contract. As seen in Fig. 4, you can check the transaction details and other status information, including any errors at the bottom of Remix (called Remix Console area). Click on the ∨ icon next to `Debug` button

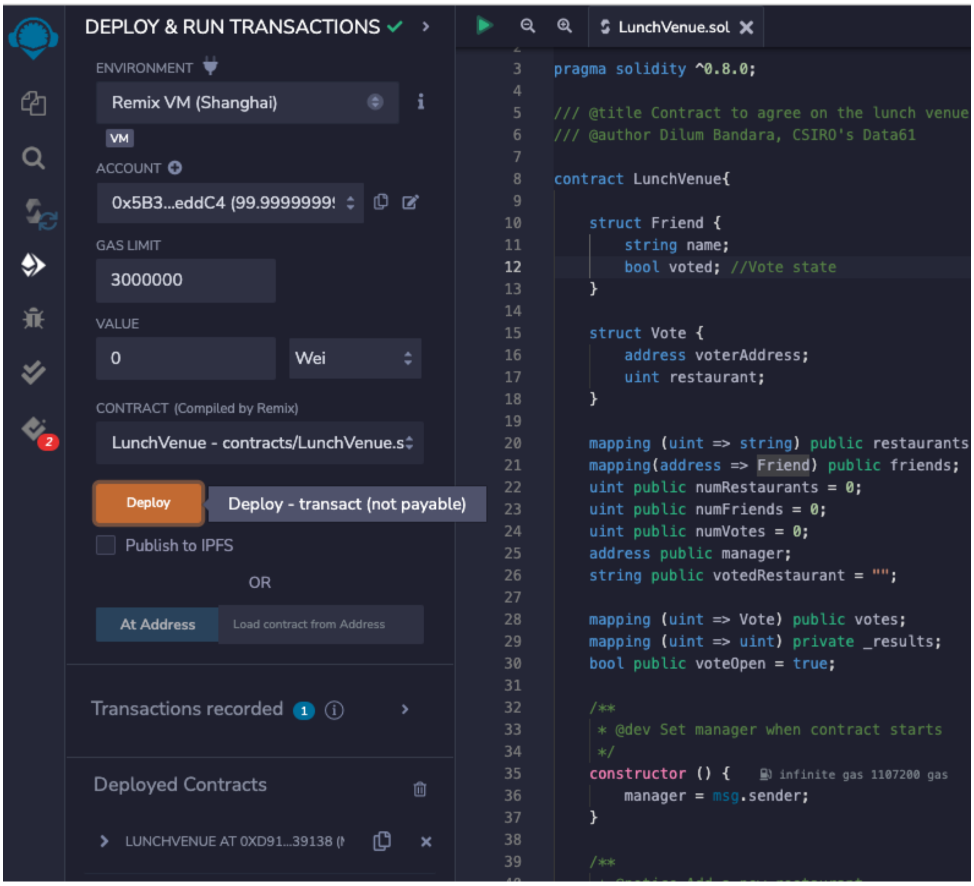

Figure 3: Deployment settings.

at the bottom left of the screen. Note values such as `status`, `contract address`, `transaction cost`, and `execution cost`. In the next section, we interact with our contract.

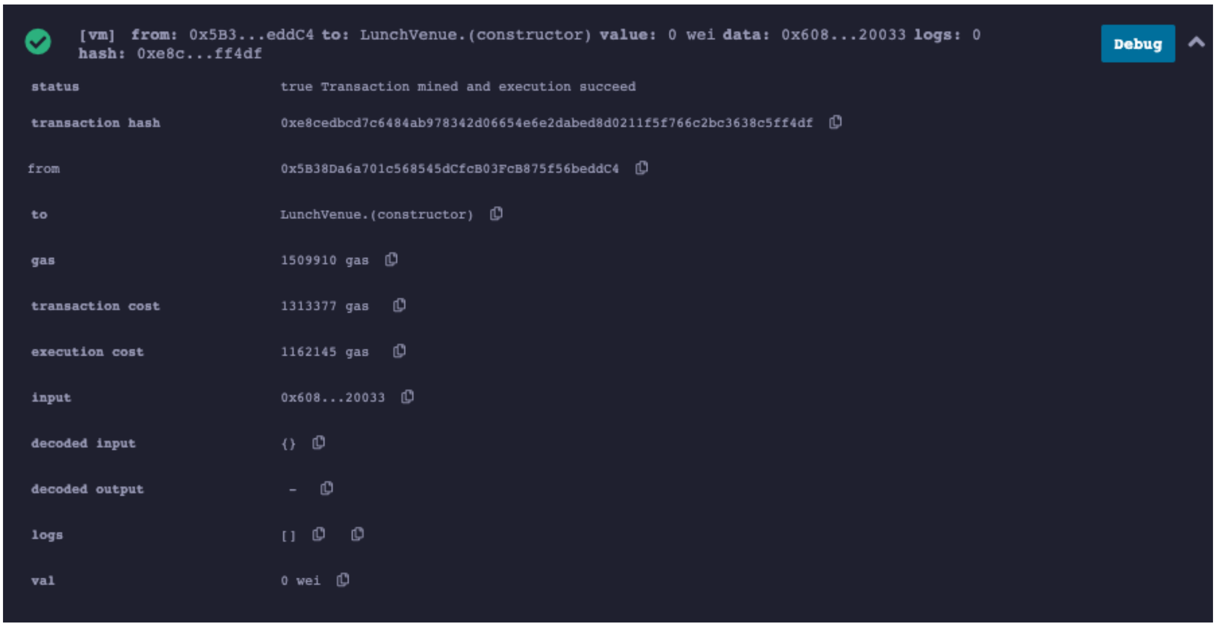

Figure 4: Details of the transaction that deployed the contract.

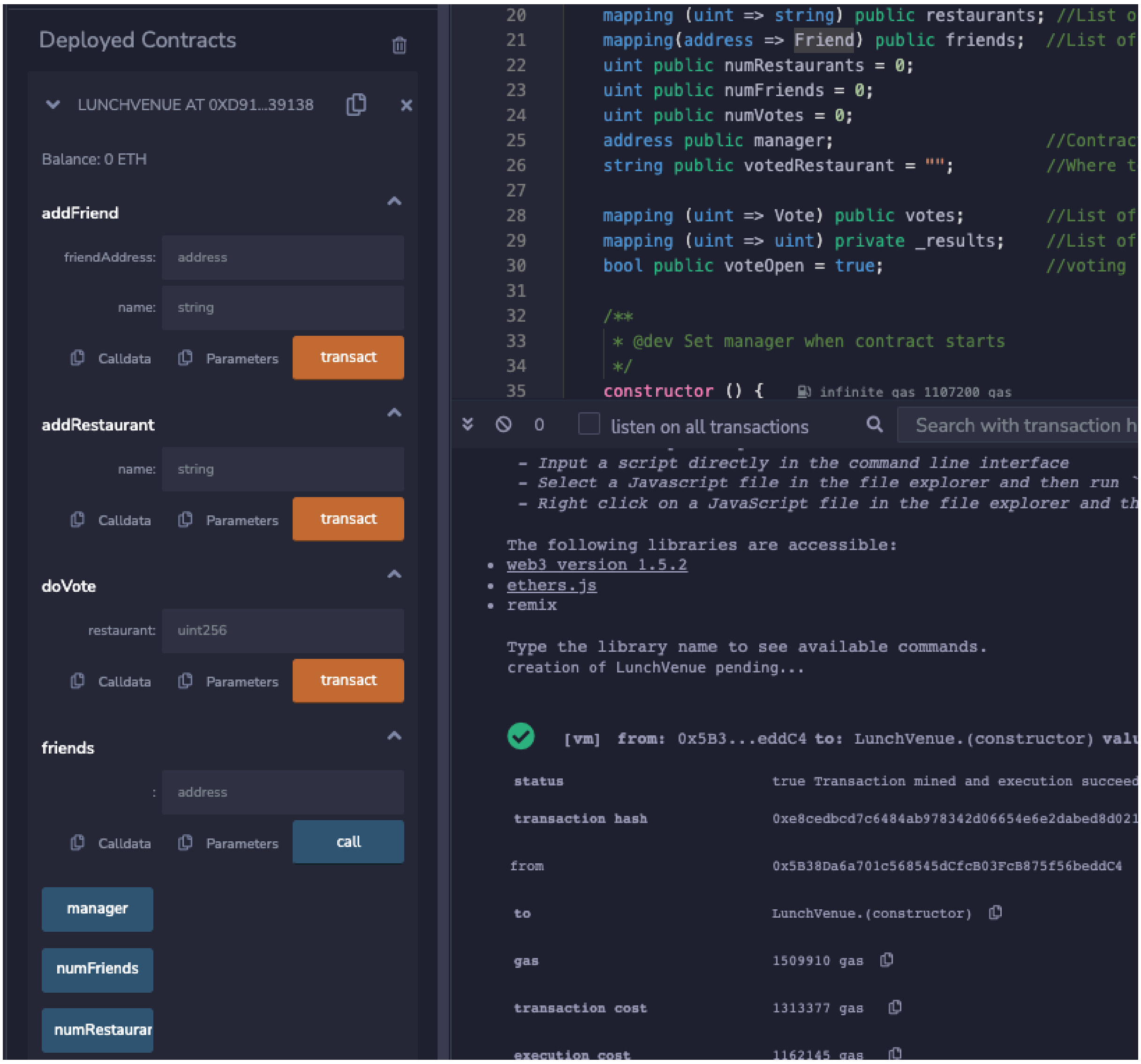

Figure 5: Interacting with the contract.

5 Manual Testing

Now that you have deployed the LunchVenue contract onto the Remix JavaScript VM, let us test it manually via Remix to make sure it works as intended.

Step 9. As seen in Fig. 5, we can interact with the deployed contract using the functions under `Deployed Contracts`. Expand the user interface by clicking on the > symbol where it says `LUNCHVENUE AT 0X...`

Those buttons can be used to generate transactions to invoke respective functions. For example, by clicking on the `manager` button, we can see that manager's address is set to the address of the account we used to deploy the smart contract. The address selected in the `ACCOUNT` drop-down (scroll up to see the drop-down list) is the one we used to deploy the contract. When we click the button, Remix issues a transaction to invoke the getter function that returns the manager's address. The respective transaction will appear on the boom of the screen. Getter functions are read-only (when the compiler generates them, they are marked as `view` only functions), so they are executed only on the node where the transaction is submitted. A read-only transaction does not consume gas as it is not executed across the blockchain network.

Similarly, check the `numFriends`, `numVotes`, and `voteOpen` variables by clicking on the respective buttons.

Step 10. To add yourself as a voter/friend, click on the ∨ icon next to the `addFriend` button, which should show two textboxes to enter input parameters. As the address selected in the `ACCOUNT` drop-down list was used to deploy the contract, let us consider that to be your address. Copy this address by clicking on the icon with two documents. Then paste it onto the `friendAddress:` textbox. Enter your name in the *name:* textbox. Then click the `transact` button.

This generates a new transaction, and you can check the transaction result in the Remix Console area. Note that `decoded output` attribute in Remix Console indicates the function returned the number of friends in the contract as one. Alternatively, the getter function provided by the `numFriends` button can be used to verify that a friend is added to the contract. We can also recall details of a friend by providing his/her address to the `friends` getter function.

Step 11. Go to the `ACCOUNT` drop-down list and select any address other than the one you used to deploy the contract. Copy that address. Return to the `addFriend` function and fill up the two textboxes with the copied address and a dummy friend name. Then click the `transact` button.

This transaction should fail. Check Remix Console area for details. While you cannot find the reason for failure (more on this later), you will see that the transaction still got mined and gas was consumed. The transaction failed due to the access control violation where the transaction's `from` address (i.e., `msg.sender`) did not match the manager's address stored in the contract (see lines 123-126).

Step 12. Let us add another friend as a voter. Go to the `ACCOUNT` drop-down list and copy the second address. After copying, reselect the address used to deploy the contract from the drop-down. Return to the `addFriend` function and paste the address we copied to `friendAddress:` textbox and enter the friend's name. Click the `transact` button. This should generate a new transaction. Check the transaction details, as well as make sure that `numFriends` is now increased to two.

Repeat this step three more times to add a total of five voters. Each time make sure to copy a different address from the `ACCOUNT` drop-down list.

Step 13. Next, we add restaurants. Expand the `addRestaurant` function by clicking on the ∨ icon. Enter a restaurant name and then click the `transact` button. Check the transaction details on the Remix Console. Use `numRestaurants` and `restaurants` getter functions to make sure the restaurant is successfully added. Also, note the difference in gas consumed by `addRestaurant` and `addFriend` functions.

Repeat this step once or twice to add total of two to three restaurants.

Step 14. It is time to vote. Let us first vote as a friend. Go to the `ACCOUNT` drop-down list and select the second address. Expand the `doVote` function. Enter one of the restaurant numbers into the `restaurant:` textbox. If you do not remember a restaurant's number, use `restaurants` getter function to find it. Then click the `transact` button.

You can check the function's output under `decoded output` in Remix Console. A successful vote should be indicated by `true`.

Step 15. This time try to vote for a restaurant that does not exist, e.g., if you added three restaurants try voting for restaurant number four. While the transaction will be successful you will see that `decoded output` is set to `false` indicating that vote is invalid.

Step 16. Next, try to vote from an address that is not in the friend list. Go to the `ACCOUNT` drop-down list and select an address that you did not register as a voter. Return to `doVote` function and then vote for a valid restaurant number. While the transaction will be successful you will see that `decoded output` is again set to `false` indicating that vote is invalid.

Step 17. Go to the `ACCOUNT` drop-down list and select an address that you registered as a friend. Return to `doVote` function and then vote for a valid restaurant number. Keep voting from different valid addresses to valid restaurants.

Once the quorum is reached, the contract will select the more voted restaurant as the lunch venue. The selected venue can be found by calling the `votedRestaurant` getter function.

Try to issue another vote and see what happens to the transaction.

6 Creating and Funding an Account

Now that the `LunchVenue` contract is working as expected, let us deploy it to the Ethereum Sepolia testnet. We need a digital wallet to create and issue Ethereum transactions. Also, we need test Ether to deploy and interact with the contract.

In this lab, we use MetaMask, a browser-based, easier-to-use, and secure way to connect to blockchain-based applications. Once the account is created, we will fund it using an already deployed faucet smart contract on the test network. We also use Etherscan.io – a blockchain explorer or search engine for Ethereum data – to check the transaction details.

Step 18. Visit `https://metamask.io/` and install the browser extension (it works on Chrome, Firefox, Edge, Opera, and Brave browsers).

Step 19. Once installed, click on the `Get Started` button. A first-time user must create a new wallet. Click on the `Create a new wallet` button. Read and agree to the policy, enter a new password twice, and click `Create a new wallet`. This will generate a 12-word or higher `Secret Backup Phrase` (aka mnemonic) to recover your wallet in case of an error. Save the mnemonic in a secure location. You are required to confirm the `Secret Backup Phrase` too.

Finally, MetaMask creates a new account with associated public and private key pairs. Your 160-bit address is derived from the *public key*, and the *private key* is used to sign transactions. Your address will appear as a long hexadecimal string with the prefix `0X` at the top of the MetaMask plugin.

Click `Copy to clipboard` to copy your address and see it on any text editor (you may have to move the mouse pointer to where it says `Account 1`). You can also get your address as a QR code or even update your account name and export your private key using the `Account details` button. Notice that your current account balance is `0 ETH`.

Step 20. Next, let us fund our account with test Ether. For this, we use a faucet smart contract that donates Ether. Because we will use the Sepolia testnet to deploy our smart contract, from the drop-down list at the top of MetaMask, select `Sepolia test network` (see Fig. 6).

If the `Sepolia test network` is not visible, click on `Show/hide test networks`. Then find the `Show/hide test networks` slider and set it to `ON`.

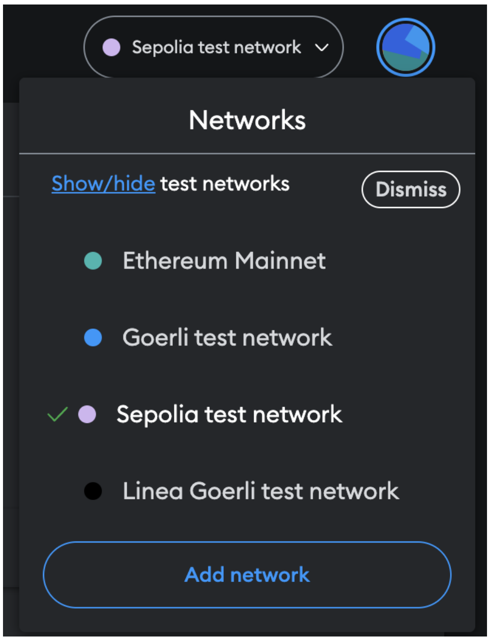

Figure 6: Selecting a testnetwork.

Step 21. Sepolia testnet has a couple of faucets (see `https://ethereum.org/en/developers/docs/networks/#sepolia`). Try one of the following faucets:

- Sepolia PoW Faucet at `https://sepolia-faucet.pk910.de/` – Copy your MetaMask `Account 1` address into the textbox that says `Please enter ETH address or ESN name`. Prove you are a human by completing the captcha. Next, click on the `Start Mining` button. Wait until you accumulate at least 0.05 ETH, which could take several minutes. Then click on the `Stop mining and claim reward` button. Finally, click on the `Claim rewards` to claim your test Ether. This should create a new transaction to fund your account. Click on the transaction ID link, which will take you to `https://etherscan.io`.
- Infura Sepolia faucet at `https://www.infura.io/faucet/sepolia` – Copy your MetaMask `Account 1` address into the textbox and click on `LOGIN AND RECEIVE ETH`. You will be required to create an Infura account. It is not a bad idea, because Infura is useful for blockchain developers as it provides connectivity and APIs to several public blockchains. You will get a popup with the title `TRANSACTION SENT!`. Click on the `VIEW ON BLOCK EXPLORER` link which will take you to `https://etherscan.io`.

As seen in Fig. 7, on Etherscan, you can see details of the transaction such as Transaction Hash, Status, From, To, and Value. For a few tens of seconds, the transaction `Status` may appear as `Pending`. Once the transaction is included in a block, the `Status` changes to `Success` and additional details such as Block, Timestamp, and Transaction Fee will appear. Click on the `Click to see More` link to see further details of the transaction.

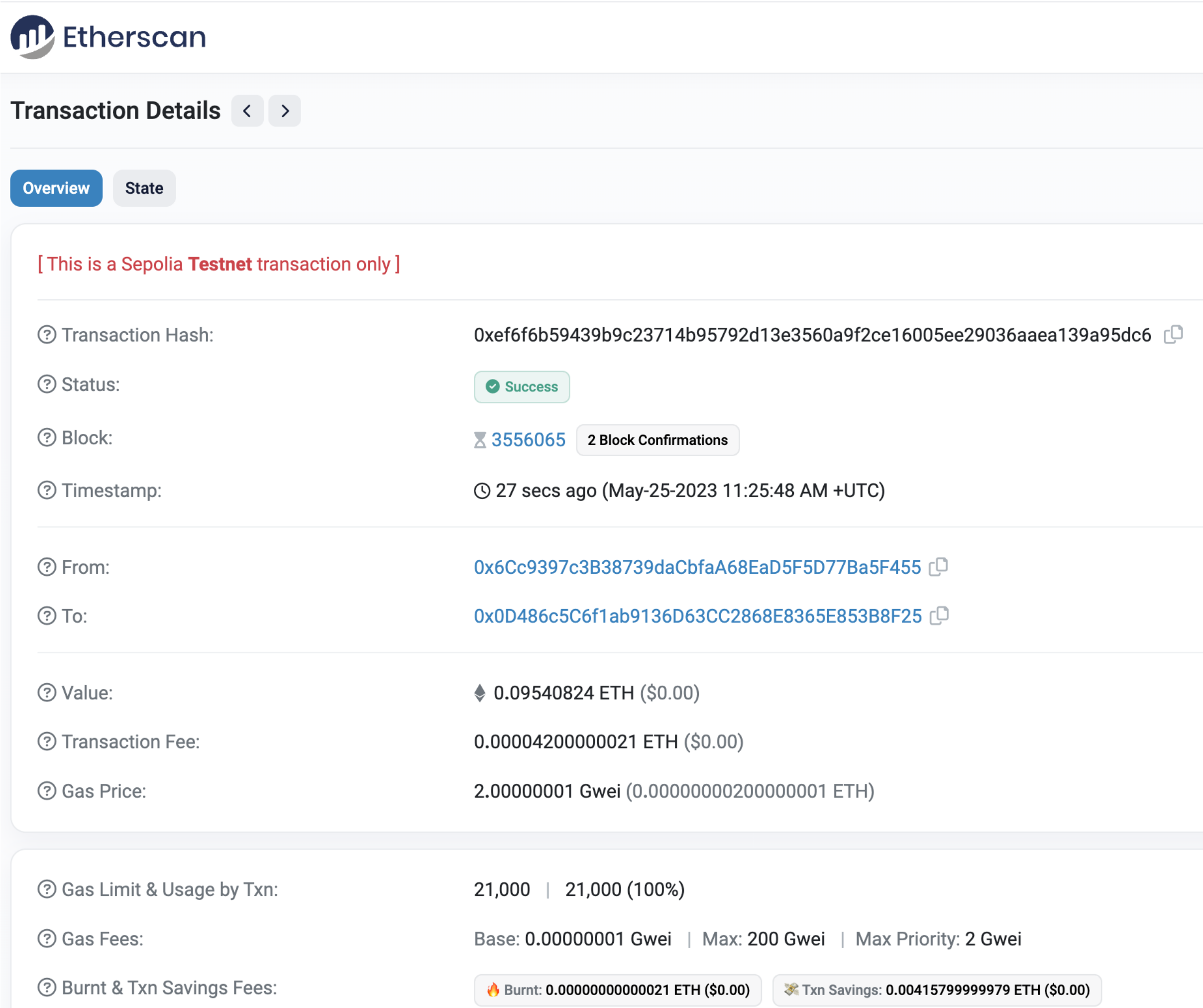

Figure 7: Transaction details.

Return to MetaMask. You will now see a small amount of ETH in your account. This should be sufficient to deploy and issues transactions to our token contract.

Step 22. Because our contract requires interaction among multiple accounts, create at least two other accounts for testing. Click on the circle in the top-right corner of MetaMask and then click on `Create account`. Follow the instructions to create a new account. Copy this address and return to one of the Sepolia faucets, enter the new address, mine, and claim your reward like in Step 21.

Alternatively, you can work with other students who are taking this class to vote from each other's addresses.

7 Deploying Smart Contract on Testnet

Step 23. Return to the Remix IDE. In the `Deploy & run transactions` pane change the `Environment` drop-down list to `Injected Provider - Metamask`. If you do not see MetaMask, you will need to reload Remix IDE.

The first time, MetaMask will pop up asking you to connect with Remix. Make sure your address is set as the `Account`. Follow the instructions to complete linking MetaMask with Remix IDE.

Step 24. Click on the `Deploy`, button to deploy the contract. MetaMask will pop up again, asking you to confirm the transaction to deploy the contract.

You will see that MetaMask has already set a transaction fee. If it is set to 0, change the value by clicking on the `Edit` link. This will list some suggested gas prices based on the expected time to include a transaction in a block. Then click on the `Confirm` button.

This will generate and send a transaction to the Sepolia testnet. You can find a link to Etherscan with the transaction ID on the Remix Console. Click on the link and wait till the transaction is included in a block. When the `Status` is marked as `Success`, your smart contract is successfully deployed onto the test network. Carefully go through the transaction parameters. Note down the `To` address, which is the address of our contract. If you lose it, it is impossible to access the contract.

If the `Success` status is marked as `Failed`, check the error messages on the Remix Console. Do the needful to fix the error and attempt to redeploy the contract.

Congratulations on deploying your first smart contract to a global network!

Step 25. Let us now interact with the smart contract on the testnet and validate its functionality. Enter the contract address in the textbox near the `At Address` button, if not already populated. Once the button is clicked, like Fig. 5, Remix will populate the UI with a list of buttons and textboxes to interact with the newly deployed contract.

Step 26. Repeats the tests from Steps 9 to 17. You would need to iterate between multiple accounts created on MetaMask.

Alternatively, you can share your LunchVenue contract address on the test network with other students in the class and get them to issue transactions to your contract.

8 Unit Testing

In this section, we automatically test our smart contract using Remix's unit testing feature. A *unit test* is a software test that focuses on components of a software product to determine whether they are fit for use. First, we write test cases. Then every time we make code changes, we can check whether our changes break the rest of the code by automatically checking whether test cases are satisfied (of course, changes may require new or modified test cases).

Step 27. Click on the `Solidity unit testing` option on the left (look for the double-check icon). If the icon is not visible, you must activate it from the Remix plugin manager. Click on the icon at the bottom left. Search for Solidity Unit Testing plugin by typing "test" in the search box, and then load it up by clicking on the `Activate` button.

Step 28. The unit testing configuration area should be like Fig. 8. Make sure `tests` is set as the `Test directory:`. Else, create the `tests` folder by clicking on the `Create` button. Then click on the `Generate` button to generate a sample Solidity test file. Usually, the name of the test file reflects our contract name and has a suffix of `_test`. This file contains the minimum code to run unit tests.

Click on the `How to use...` button. Read through the *Unit Testing Plugin* web page and other pages in the section to get an idea about how to perform unit testing with Remix.

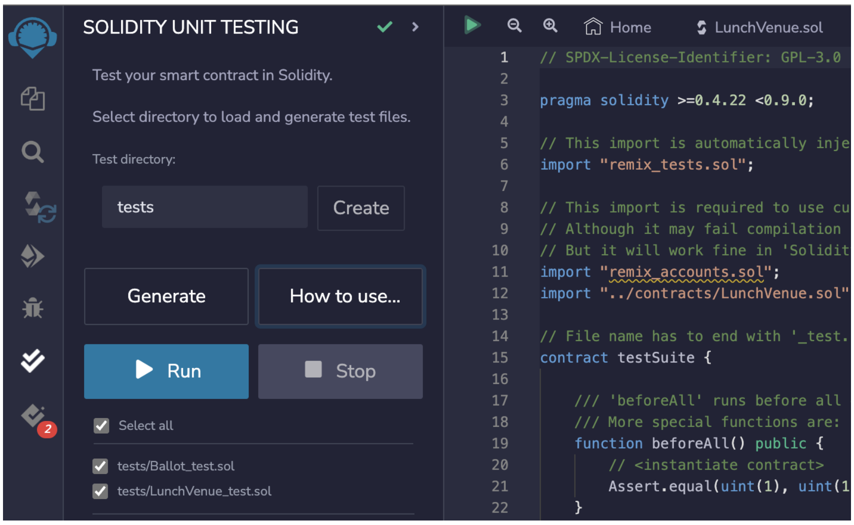

Figure 8: Unit test options.

Step 29. Let us write the unit tests for our contract. Edit the `LunchVenue_test.sol` file in the `tests` folder to include the following unit test code. You will have to remove some of the boilerplate code from Remix.

In line 6, we import the `remix_tests.sol` file, which includes a set of utility functions for testing contracts. The `remix_accounts.sol` file (included in line 11) gives access to ten test accounts to emulate the behaviour of Ethereum accounts. Line 12 imports our `LunchVenue` contract to be tested. In line 16, we inherit the `LunchVenue` contract to test its functionality. This is required as we want to emulate user behaviour.

`beforeAll` function runs before all the tests; hence, it can be used to set states needed for testing. Between lines 19-23, we create a set of variables for test user accounts. Their values are assigned to test accounts between lines 29-33.

As per the configuration in `remix_accounts.sol`, account at zero index (i.e., `account-0`) is the default account. Therefore, it is automatically used to deploy the contract during testing. Consequently, it becomes our contact's owner and the `manager` variable is set to this account. `acc0` to `acc3` are used as friends who could vote for a lunch venue. Note that `account-0` is a label that can be used to set the transaction sender, whereas `acc0` is the respective variable.

`managerTest` (lines 38-40) is our first unit test that validates whether the default account is set as the manager of the contract. An `Assert.equal` function compares whether its first and second arguments are the same. Otherwise, it throws an exception with the error message given in the third argument. For example, in line 39, we check whether the `manager` is set to `acc0`. If not, it will throw an exception with the error message "Manager should be acc0"

`setRestaurant` (lines 44-47) test case adds two restaurants where we expect the smart contract to return the number of restaurants available to vote for each addition.

In the `setRestaurantFailure` test case (lines 51-64), we try to add another lunch venue. `#sender : account-1` in line 50 indicates that we are calling the function while setting `account-1` as the `msg.sender` of the transaction. This is a convenient feature Remix unit testing provides to issue transactions as originating from different addresses. This test case should fail, as only the manager is allowed to add a restaurant. However, in unit testing, we are checking for expected behaviour, which in this case is a failed transaction. Hence, without letting the test case fail, we can capture the failure as the accepted behaviour using a `try-catch` block.

In programming, `try` defines a block of statements that may throw an exception. When a specific type of exception occurs, a `catch` block catches the exception enabling us to handle the error within the program without crashing it. Therefore, when we execute line 54, we should reach line 56 as the EVM will throw an error with the reason. We check for this behaviour using `Assert.equal` in line 58.

If we get some other error or the transaction is successful, the unit test should fail at line 60, 62, or 55, respectively.

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  pragma solidity >=0.8.00 <0.9.0;
4
5  // This import is automatically injected by Remix
6  import "remix_tests.sol";
7
8  // This import is required to use custom transaction context
9  // Although it may fail compilation in 'Solidity Compiler' plugin
10 // But it will work fine in 'Solidity Unit Testing' plugin
11 import "remix_accounts.sol";
12 import "../contracts/LunchVenue.sol";
13
14 // File name has to end with '_test.sol', this file can contain more than one testSuite
       contracts
15 /// Inherit 'LunchVenue' contract
16 contract LunchVenueTest is LunchVenue {
17
18     // Variables used to emulate different accounts
19     address acc0;
20     address acc1;
21     address acc2;
22     address acc3;
23     address acc4;
24
25     /// 'beforeAll' runs before all other tests
26     /// More special functions are: 'beforeEach', 'beforeAll', 'afterEach' & 'afterAll'
27     function beforeAll() public {
28         // Initiate account variables
29         acc0 = TestsAccounts.getAccount(0);
30         acc1 = TestsAccounts.getAccount(1);
31         acc2 = TestsAccounts.getAccount(2);
32         acc3 = TestsAccounts.getAccount(3);
33         acc4 = TestsAccounts.getAccount(4);
34     }
35
36     /// Check manager
37     /// account-0 is the default account that deploy contract, so it should be the
           manager (i.e., acc0)
38     function managerTest() public {
39         Assert.equal(manager, acc0, 'Manager should be acc0');
40     }
41
42     /// Add restuarant as manager
43     /// When msg.sender isn't specified, default account (i.e., account-0) is the sender
44     function setRestaurant() public {
45         Assert.equal(addRestaurant('Courtyard Cafe'), 1, 'Should be equal to 1');
46         Assert.equal(addRestaurant('Uni Cafe'), 2, 'Should be equal to 2');
47     }
48
49     /// Try to add a restaurant as a user other than manager. This should fail
50     /// #sender: account-1
51     function setRestaurantFailure() public {
52         // Try to catch reason for failure using try-catch . When using
53         // try-catch we need 'this' keyword to make function call external
54         try this.addRestaurant('Atomic Cafe') returns (uint v){
55             Assert.notEqual(v, 3, 'Method execution did not fail');
56         } catch Error(string memory reason) {
57             // Compare failure reason, check if it is as expected
58             Assert.equal(reason, 'Can only be executed by the manager', 'Failed with
                   unexpected reason');
59         } catch Panic(uint /* errorCode */) { // In case of a panic
60             Assert.ok(false, 'Failed unexpected with error code');
61         } catch (bytes memory /*lowLevelData*/) {
62             Assert.ok(false, 'Failed unexpected');
63         }
64     }
65
66     /// Set friends as account-0
67     /// #sender doesn't need to be specified explicitly for account-0
68     function setFriend() public {
```

```
69         Assert.equal(addFriend(acc0, 'Alice'), 1, 'Should be equal to 1');
70         Assert.equal(addFriend(acc1, 'Bob'), 2, 'Should be equal to 2');
71         Assert.equal(addFriend(acc2, 'Charlie'), 3, 'Should be equal to 3');
72         Assert.equal(addFriend(acc3, 'Eve'), 4, 'Should be equal to 4');
73     }
74 
75     /// Try adding friend as a user other than manager. This should fail
76     function setFriendFailure() public {
77         try this.addFriend(acc4, 'Daniels') returns (uint f) {
78             Assert.notEqual(f, 5, 'Method execution did not fail');
79         } catch Error(string memory reason) { // In case revert() called
80             // Compare failure reason, check if it is as expected
81             Assert.equal(reason, 'Can only be executed by the manager', 'Failed with
                   unexpected reason');
82         } catch Panic( uint /* errorCode */) { // In case of a panic
83             Assert.ok(false , 'Failed unexpected with error code');
84         } catch (bytes memory /*lowLevelData*/) {
85             Assert.ok(false, 'Failed unexpected');
86         }
87     }
88 
89     /// Vote as Bob (acc1)
90     /// #sender: account-1
91     function vote() public {
92         Assert.ok(doVote(2), "Voting result should be true");
93     }
94 
95     /// Vote as Charlie
96     /// #sender: account-2
97     function vote2() public {
98         Assert.ok(doVote(1), "Voting result should be true");
99     }
100 
101   /// Try voting as a user not in the friends list. This should fail
102     /// #sender: account-4
103     function voteFailure() public {
104         Assert.equal(doVote(1), false, "Voting result should be false");
105     }
106 
107     /// Vote as Eve
108     /// #sender: account-3
109     function vote3() public {
110         Assert.ok(doVote(2), "Voting result should be true");
111     }
112 
113     /// Verify lunch venue is set correctly
114     function lunchVenueTest() public {
115         Assert.equal(votedRestaurant, 'Uni Cafe', 'Selected restaurant should be Uni Cafe
                ');
116     }
117 
118     /// Verify voting is now closed
119     function voteOpenTest() public {
120         Assert.equal(voteOpen, false, 'Voting should be closed');
121     }
122 
123     /// Verify voting after vote closed. This should fail
124     function voteAfterClosedFailure() public {
125         try this.doVote(1) returns (bool validVote) {
126             Assert.equal(validVote, true, 'Method execution did not fail');
127         } catch Error(string memory reason) {
128             // Compare failure reason, check if it is as expected
129             Assert.equal(reason, 'Can vote only while voting is open.', 'Failed with
                   unexpected reason');
130         } catch Panic( uint /* errorCode */) { // In case of a panic
131             Assert.ok(false , 'Failed unexpected with error code');
132         } catch (bytes memory /*lowLevelData*/) {
133             Assert.ok(false, 'Failed unexpectedly');
134         }
135     }
136 }
```

When we call `this.addRestaurant` in line 54, we are issuing the transaction as "this" contract. The keyword `this` refers to the contract itself, i.e., the `LunchVenue` unit test contract. Therefore, `msg.sender` is set to `this` contract's address, not to the manager's address which is set to `account-0`. Because the `msg.sender` and `manager` are not the same, line 124 in the `LunchVenue` contract throws an exception with the error message "Can only be executed by the manage". We catch this exception in line 58 in the unit test file as the expected result. Even if you change line 50 to `#sender: account-0` the test case will not fail as `this` always refers to the contract's address. In fact, line 50 can be removed as the sender's address we set is immaterial in this test case. It is just added to help you understand what is going on (e.g., `setFriendFailure` and `voteAfterClosedFailure` test cases do not set the sender address).

`setFriend` and `setFriendFailure` test cases try to add friends that can vote for a restaurant. In `vote` and `vote2` test cases, we vote for a restaurant as Bob and Charlie, respectively.

Because `account-4` is not in the friends list, the `voteFailure` test case (lines 103-105) `doVote` should return `false`. Next, we vote as Eve (lines 109-111). As the minimum number of votes is received, the smart contract should select *Uni Cafe* as the highest-voted restaurant and disable further voting. `lunchVenueTest` and `voteOpenTest` test cases verify this behaviour. Finally, between lines 124 and 135, we make sure no more votes can be cast once the lunch venue is decided.

Step 30. To run our test cases, go to the `Solidity unit testing` pane. Select `LunchVenue_test.sol` from the set of checkboxes. Then click on the `Run` button (see Fig. 8). You should see an output like Fig. 9.

All tests should be successful. If not, check the error messages, apply necessary fixes, and retry.

Ideally, unit testing must be performed before you test it either on Remix JavaScript VM or on a test network. However, this lab presented topics in a pragmatic order to make it easier to follow and retain the motivation than bombarding you with more Solidity code. Therefore, in Project 1 and 2 make sure to complete your unit testing before any deployment.

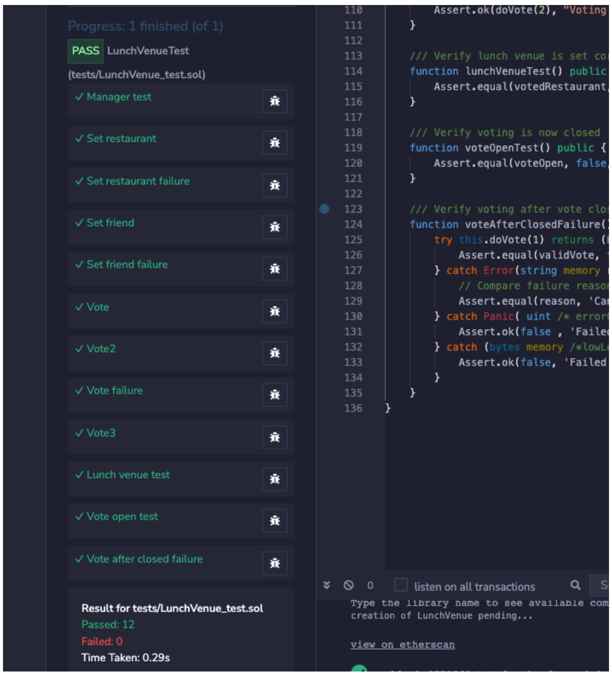

Figure 9: Unit test results.

9 Project 1 - Extending the Smart Contract

While our smart contract works, it has a couple of issues. For example:

1. A friend can vote more than once. While this was handy in testing our contract, it is undesirable as one could monopolise the selected venue. Ideally, we should record a vote for a given address only once.
2. The same restaurant and friend can be added multiple times leading to similar issues.

3. While the contract is stuck at the `doVote` function, other functions can still be called. Also, once the voting starts, new venues and friends can be added, making the whole process chaotic. In a typical voting process, voters are clear about who would vote and what to vote on before voting starts to prevent any disputes. Hence, a good vote process should have well-defined create, vote open, and vote close phases.

4. If the quorum is not reached by lunchtime, no consensus will be reached on the lunch venue. Hence, the contract needs a timeout. However, the wallclock time on a blockchain is not accurate due to clock skew. Therefore, the timeout needs to be defined as a block number.

5. There is no way to disable the contract once it is deployed. Even the manager cannot do anything to stop the contract in case the team lunch has to be cancelled.

6. Gas consumption is not optimised. More simple data structures may help to reduce transaction fees.

7. Unit tests do not cover all possible cases, e.g., we do not check the `doVote` function with an invalid restaurant number.

Your Project 1 task is to address these issues by improving the smart contract and unit tests.

Step 31. Update the `LunchVenue` smart contract to satisfy at least five of the above-listed weakness. Save it as a separate `.sol` file. Do not modify function definitions unless essential. Also, clearly mention which weaknesses you address and how you address them. These could be added as comments to your code. You may need to look into more Solidity functions to resolve some of the issues.

Step 32. Create a new unit test file for the updated contract. In addition to fully testing the original contract, add at least four other test cases to validate the new functionality. Proceed with Step 24 for the updated contract too. That way, we can check your work (code and transactions), in addition to us deploying a new instance of your contract.

10 Project Submission

You are required to submit the following as a single .zip or .tar.gz file:

Deliverable	**Points (15 in total)**
Source code of original LunchVenue.sol	1
Source code of original LunchVenue_test.sol	1
Source code of updated .sol file to fix at least 5 weaknesses	8
Source code of updated _test.sol test file	4
2 addresses of above smart contracts deployed on Sepolia as addresses.txt	1

You should have the following structure of directories in your submitted .zip or .tar.gz file (DO NOT include the root directory):

- contracts/
  - LunchVenue.sol
  - LunchVenue_updated.sol
- tests/
  - LunchVenue_test.sol
  - LunchVenue_updated_test.sol
- addresses.txt

The contracts should be written with Solidity version **0.8.x**.
These need to be submitted by the deadline given on the course Moodle page.

- **The standard late penalty applies as per the UNSW assessment implementation procedure. The late penalty is a per-day (not hourly) mark reduction equal to 5% of the max assessment mark, for up to 5 days. Zero marks after 5 days. All days, not just Monday-Friday, are included in the days late. See course Moodle page for examples on how penalty is calculated.**
- **Plagiarism checker will be used to analyze the submitted code and answer for open question (changing the name of state variables will not help). The UNSW has an ongoing commitment to fostering a culture of learning informed by academic integrity. All the UNSW staff and students have a responsibility to adhere to this principle of academic integrity. Plagiarism undermines academic integrity and is not tolerated at the UNSW.**

References

[1] L. Lessig. (2000) Code is law. [Online]. Available: https://harvardmagazine.com/2000/01/code-is-law-html

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] V. Buterin. (2014) A next-generation smart contract and decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger: Byzantium version," *Ethereum project yellow paper*, Mar. 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[5] Ethereum. (2021) Solidity. [Online]. Available: https://solidity.readthedocs.io/