Blockchain technology lab manual

Computer science (Malla Reddy Group of Institutions)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

MALLA REDDY ENGINEERING COLLEGE AND MANAGEMENT SCIENCES

(APPROVED BY AICTE NEW DELHI AND AFFILIATED TO JNTUH HYDERABAD)

Kistapur, Medchal, Rangareddy – 501401

B.Tech III YEAR I SEM (R18)

(2021-2022)

BLOCKCHAIN TECHNOLOGY

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

1. Setup Metamask in the System and Create a wallet in the Metamask with Test Network

Aim: To set up Metamask in the system and create a wallet in Metamask with Test Network.

Requirements:

- A computer with a modern web browser (Google Chrome, Firefox, Brave, etc.)
- Internet connectivity
- Metamask browser extension
- A supported test network (Rinkeby, Ropsten, Kovan, etc.)
- Test network Ether for transactions (available for free from a faucet)

Procedure:

1. Install the Metamask browser extension from the official website (https://metamask.io/).
2. Once installed, click on the Metamask icon in your browser toolbar and select "Create a Wallet."
3. Follow the prompts to create a new wallet with a secure password. Be sure to write down your secret backup phrase and store it in a safe place.
4. After creating your wallet, you will be prompted to choose a network. Select the test network you wish to use (Rinkeby, Ropsten, Kovan, etc.).
5. If you don't have any test network Ether, you can get some for free from a faucet. To do this, simply search for "test network Ether faucet" online, and you will find a number of options.
6. Once you have your test network Ether, you can use Metamask to make transactions on the test network.

Source Code:

N/A

Output:

After following the above procedure, you will have successfully set up Metamask in your system and created a wallet with a test network. You will be able to send and receive test network Ether and interact with smart contracts on the test network.

Conclusion:

Metamask is a popular wallet and browser extension that makes it easy to interact with decentralized applications (dApps) on the Ethereum network. By setting up Metamask with a test network, you can experiment with dApps and smart contracts without risking real Ether.

Result:

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

By following this guide, you should now have a functional Metamask wallet with a test network. You can now start experimenting with dApps and smart contracts, and sending and receiving test network Ether.

2.Create multiple accounts in metamask and perform the balance transfer between the accounts and describe the transaction specifications

Aim: To create multiple accounts in Metamask and perform a balance transfer between the accounts, and describe the transaction specifications.

Requirements:

- A computer with a modern web browser (Google Chrome, Firefox, Brave, etc.)
- Internet connectivity
- Metamask browser extension
- Test network Ether for transactions (available for free from a faucet)

Procedure:

1. Install the Metamask browser extension from the official website (https://metamask.io/).
2. Once installed, click on the Metamask icon in your browser toolbar and select "Create a Wallet."
3. Follow the prompts to create a new wallet with a secure password. Be sure to write down your secret backup phrase and store it in a safe place.
4. After creating your wallet, you will be taken to your Metamask dashboard. Click on the account icon in the top right corner and select "Create Account" to create a new account.
5. Repeat step 4 to create as many accounts as you need.
6. To transfer funds between accounts, select the account you wish to send Ether from and click "Send."
7. Enter the recipient's address (which can be copied from the recipient's account in Metamask) and the amount of Ether you wish to send.
8. Review the transaction details, including gas fees, and click "Confirm" to initiate the transfer.

Source Code:

N/A

Output:

After following the above procedure, you will have created multiple accounts in Metamask and performed a balance transfer between them. The transaction specifications, including the transaction hash, sender and recipient addresses, and amount transferred, can be viewed on the blockchain explorer for the test network you used.

Conclusion:

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

Metamask makes it easy to create and manage multiple accounts and transfer funds between them. Transactions on the Ethereum network are recorded on the blockchain, and transaction specifications can be viewed on blockchain explorers like Etherscan.

Result:

By following this guide, you should now be able to create and manage multiple accounts in Metamask and perform balance transfers between them. You can experiment with different transaction types and amounts to gain a better understanding of how the Ethereum network works.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

3. Setup the Ganache Tool in the system

Aim: To set up the Ganache tool in the system.

Requirements:

- A computer with a modern web browser (Google Chrome, Firefox, Brave, etc.)
- Internet connectivity
- Ganache tool installer (available for free from the official website)

Procedure:

1. Download the Ganache tool installer from the official website (https://www.trufflesuite.com/ganache).
2. Open the downloaded file and follow the prompts to install Ganache on your system.
3. Once installed, open the Ganache application.
4. By default, Ganache will create a new workspace with ten accounts pre-populated with test Ether.
5. You can configure the workspace settings, such as the number of accounts and their starting balance, by clicking on the gear icon in the top right corner of the window.
6. Once you have configured the workspace to your liking, you can use it to test and deploy smart contracts on the Ethereum network.

Source Code:

N/A

Output:

After following the above procedure, you will have successfully set up the Ganache tool in your system. You will be able to create and manage workspaces, accounts, and test Ether balances, and use them to test and deploy smart contracts on the Ethereum network.

Conclusion:

Ganache is a popular tool for local development and testing of smart contracts on the Ethereum network. By setting up Ganache in your system, you can create custom workspaces and test various scenarios before deploying your smart contracts on the live network.

Result:

By following this guide, you should now have Ganache set up in your system and be able to create custom workspaces for testing and deploying smart contracts.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

4. Create a custom RPC network in Metamask and connect it with Ganache tool and transfer the ether between ganache accounts.

Aim: To create a custom RPC network in Metamask and connect it with Ganache tool to transfer ether between Ganache accounts.

Requirements:

- A computer with Metamask and Ganache installed.
- Ganache tool running and a workspace created with at least two accounts with test ether.
- Internet connectivity.

Procedure:

1. Open the Ganache tool and create a new workspace or select an existing one.
2. Note the RPC server endpoint, which is usually displayed in the workspace settings. It will be something like http://127.0.0.1:7545.
3. Open Metamask in your browser and click on the network dropdown menu on the top of the window.
4. Select "Custom RPC" from the bottom of the list.
5. In the "New Network" section, enter a name for the network and the RPC server endpoint that you noted in step 2.
6. Click "Save" to create the custom RPC network.
7. Switch to the custom RPC network by selecting it from the network dropdown menu in Metamask.
8. Click on the account dropdown menu and select one of the accounts from your Ganache workspace.
9. Click on "Send" to initiate a transaction and enter the recipient address and the amount of ether to transfer.
10. Confirm the transaction and wait for it to be confirmed on the blockchain.

Source Code:

N/A

Output:

After following the above procedure, you will have successfully created a custom RPC network in Metamask and connected it with Ganache tool to transfer ether between Ganache accounts. You will be able to switch between the custom RPC network and the other networks in Metamask and send transactions between Ganache accounts.

Conclusion:

By creating a custom RPC network in Metamask and connecting it with Ganache tool, you can test and develop smart contracts locally and transfer test ether between accounts. This

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

allows you to simulate real-world scenarios and test your smart contracts before deploying them to the live network.

Result:

By following this guide, you should now be able to create a custom RPC network in Metamask and connect it with Ganache tool to transfer ether between Ganache accounts. You can use this setup for local development and testing of smart contracts on the Ethereum network.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

5. Write a smart contract using a solidity program to perform the balance transfer from contract to other accounts

Aim: To write a Solidity smart contract that performs balance transfer from the contract to other accounts.

Requirements:

- Solidity compiler (version 0.8.0 or above)
- A development environment (such as Remix, Visual Studio Code with Solidity extension, etc.)

Procedure:

1. Open your development environment and create a new Solidity file.
2. Copy and paste the above code into the file.
3. Compile the contract using the Solidity compiler to check for any errors or warnings.
4. Deploy the contract to the Ethereum network.
5. To transfer balance from the contract to other accounts, call the `transfer` function with the recipient address and the amount to transfer as arguments.
6. To check the balance of the contract, call the `getContractBalance` function.

Source Code:

```
pragma solidity ^0.8.0;

contract BalanceTransfer {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transfer(address payable _to, uint256 _amount) public {
        require(msg.sender == owner, "Only contract owner can transfer
balance");
        require(address(this).balance >= _amount, "Insufficient balance in
contract");

        _to.transfer(_amount);
    }

    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```


Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

Output:

After deploying the contract and calling the `transfer` function with the recipient address and the amount to transfer, the balance of the contract should decrease by the transferred amount and the recipient account should receive the transferred amount.

Conclusion:

The above Solidity smart contract can be used to transfer balance from a contract to other accounts. The contract owner can call the `transfer` function to transfer balance from the contract to any account on the Ethereum network. By checking the balance of the contract using the `getContractBalance` function, the contract owner can ensure that there is sufficient balance in the contract to perform the transfer.

Result:

By following this guide, you should now be able to write a Solidity smart contract that performs balance transfer from a contract to other accounts. This can be useful for various scenarios such as paying out rewards, distributing tokens, etc.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

6. Write solidity program to perfume the exception handling.

Aim: To write a Solidity program that demonstrates exception handling.

Requirements:

- Solidity compiler (version 0.8.0 or above)
- A development environment (such as Remix, Visual Studio Code with Solidity extension, etc.)

Procedure:

1. Open your development environment and create a new Solidity file.
2. Copy and paste the above code into the file.
3. Compile the contract using the Solidity compiler to check for any errors or warnings.
4. Deploy the contract to the Ethereum network.
5. To test exception handling, call the `divide`, `multiply`, and `transfer` functions with various arguments to see how they handle exceptions.

Source Code:

```
pragma solidity ^0.8.0;

contract ExceptionHandling {
    function divide(uint256 _numerator, uint256 _denominator) public view
returns (uint256) {
        require(_denominator != 0, "Denominator cannot be zero");
        return _numerator / _denominator;
    }

    function multiply(uint256 _a, uint256 _b) public view returns (uint256)
{
        uint256 result = _a * _b;
        if (result == 0) {
            revert("Multiplication resulted in zero");
        }
        return result;
    }

    function transfer(address payable _to, uint256 _amount) public payable
{
        require(msg.value >= _amount, "Insufficient balance to transfer");
        _to.transfer(_amount);
    }
}
```

Output:

When calling the `divide` function with `_denominator` as 0, the require statement will throw an exception with the error message "Denominator cannot be zero".

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

When calling the `multiply` function with `_a` and `_b` as 0, the function will throw an exception with the error message "Multiplication resulted in zero".

When calling the `transfer` function with `msg.value` less than `_amount`, the require statement will throw an exception with the error message "Insufficient balance to transfer".

Conclusion:

Exception handling in Solidity is important for ensuring that your contracts can handle unexpected or invalid inputs. The `require` statement can be used to check for conditions that must be met in order for a function to execute, and the `revert` statement can be used to throw an exception with a custom error message.

By testing the `divide`, `multiply`, and `transfer` functions with various arguments, we can ensure that the exception handling works as expected and prevents unexpected behavior.

Result:

By following this guide, you should now be able to write a Solidity program that demonstrates exception handling. This can be useful for ensuring the safety and reliability of your smart contracts.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

7. Setup the Hyperledger Fabric Network with 2 Organizations 1 Peer Each in the system

Aim: The aim of this tutorial is to demonstrate how to set up a Hyperledger Fabric Network with 2 organizations, each having one peer.

Requirements: To complete this tutorial, you will need the following software and tools:

- Hyperledger Fabric v2.x
- Docker v17.06.2-ce or higher
- Docker Compose v1.14.0 or higher
- Git

Procedure:

Step 1: Clone the Fabric Samples Repository Open a terminal and clone the Fabric samples repository from GitHub by running the following command:

```
git clone https://github.com/hyperledger/fabric-samples.git
```

Step 2: Navigate to the Test Network Directory Change the directory to the test network folder by running the following command:

```
cd fabric-samples/test-network
```

Step 3: Generate the Crypto Materials Run the following command to generate the crypto materials:

```
./network.sh generateCrypto
```

This command will generate the crypto materials for two organizations, Orderer Org and Org1.

Step 4: Start the Network To start the network, run the following command:

```
./network.sh up createChannel -c mychannel -ca
```

This command will start the network and create a channel named mychannel.

Step 5: Join Peers to the Channel Join the peers from both organizations to the channel by running the following command:

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-
javascript/ -ccl javascript
```


Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

This command will deploy the asset transfer chaincode to the channel.

Step 6: Test the Network Test the network by running the following command to invoke the chaincode and query its state:

```
./network.sh testCC
```

Output: If everything is set up correctly, you should see the following output:

```
2021-09-30 16:35:23.546 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO
001 Chaincode invoke successful. result: status:200
payload:"{\"value\":\"100\",\"docType\":\"asset\",\"ID\":\"asset1\"}"

2021-09-30 16:35:23.577 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO
002 Chaincode invoke successful. result: status:200
payload:"{\"value\":\"100\",\"docType\":\"asset\",\"ID\":\"asset1\"}"

2021-09-30 16:35:23.607 UTC [main] main -> INFO 003 Exiting.....
```

Conclusion:

In this tutorial, we have successfully set up a Hyperledger Fabric Network with 2 organizations, each having one peer. We have also deployed a chaincode to the network and tested it.

Result:

By following this tutorial, you have learned how to set up a basic Hyperledger Fabric Network with multiple organizations and peers, and how to deploy and test a chaincode on the network.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

8 Create a channel called mychannel, carchannel in the deployed network

Aim: The aim of this tutorial is to demonstrate how to create new channels named `mychannel` and `carchannel` in a deployed Hyperledger Fabric network.

Requirements: To complete this tutorial, you will need:

- A deployed Hyperledger Fabric network
- A Peer Admin identity with sufficient privileges to create a new channel
- Fabric binaries and docker images installed
- Access to the command line

Procedure:

Step 1: Open a terminal and navigate to the **`test-network`** directory:

```
cd fabric-samples/test-network/
```

Step 2: Generate the channel artifacts for the new channel **`mychannel:`**

```
./network.sh createChannel -c mychannel
```

Step 3: Verify that the channel has been created successfully:

```
./network.sh listChannels
```

You should see the **`mychannel`** channel listed among the available channels.

Step 4: Repeat steps 2 and 3 for the **`carchannel`** channel:

```
./network.sh createChannel -c carchannel

./network.sh listChannels
```

Output:

The output of the **`listChannels`** command should look similar to the following:

```
2023-04-20 15:25:38.100 UTC [main] main -> INFO 001 Exiting.....

========= Channel ID: mychannel ===========

========= Channel ID: carchannel ===========
```

Conclusion:

In this tutorial, we have demonstrated how to create new channels named `mychannel` and `carchannel` in a deployed Hyperledger Fabric network using the `network.sh` script provided by the Fabric samples repository.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

Result:

By following this tutorial, you have learned how to create new channels in a deployed Fabric network, which can be used to segregate transactions among different organizations or applications.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

9. Take the existing Fabcar smart contract and add a new function to query the car on the basis of person name and deploy the smart contract on the Hyparledger Fabric Network

Aim: The aim of this project is to modify the existing Fabcar smart contract to add a new function that enables querying of a car based on the name of its owner. This will allow users to easily retrieve information about cars owned by a specific person.

Requirements:

To accomplish this task, we will need the following:

- Access to the existing Fabcar smart contract code
- An understanding of the Hyperledger Fabric network and how to deploy smart contracts to it
- A development environment for editing and testing the smart contract code
- Knowledge of the Solidity programming language used to write the smart contract

Procedure:

1. First, we will need to download the existing Fabcar smart contract code from the Hyperledger Fabric repository.
2. Next, we will add a new function to the code that allows querying of a car by owner name. The function will take a string parameter representing the name of the owner and will return an array of car objects that match that owner name.
3. We will test the modified smart contract locally using the Hyperledger Fabric development environment to ensure it works as intended.
4. Finally, we will deploy the modified smart contract to the Hyperledger Fabric network using the appropriate tools and verify that it functions correctly.

Source code:

Here is the modified Fabcar smart contract code with the new function for querying cars by owner name:

```
scss
pragma solidity >=0.4.22 <0.9.0;

contract FabCar {
    struct Car {
        string make;
        string model;
        uint256 year;
        string owner;
    }

    mapping (string => Car) cars;

    function addCar(string memory _carId, string memory _make, string
memory _model, uint256 _year, string memory _owner) public {
        Car memory newCar = Car(_make, _model, _year, _owner);
        cars[_carId] = newCar;
```


Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

```
    }

    function queryCar(string memory _carId) public view returns (string
memory, string memory, uint256, string memory) {
        return (cars[_carId].make, cars[_carId].model, cars[_carId].year,
cars[_carId].owner);
    }

    function queryCarsByOwner(string memory _owner) public view returns
(Car[] memory) {
        uint256 count = 0;
        for (uint256 i = 0; i < cars.length; i++) {
            if (keccak256(bytes(cars[i].owner)) ==
keccak256(bytes(_owner))) {
                count++;
            }
        }

        Car[] memory result = new Car[](count);
        uint256 index = 0;
        for (uint256 i = 0; i < cars.length; i++) {
            if (keccak256(bytes(cars[i].owner)) ==
keccak256(bytes(_owner))) {
                result[index] = cars[i];
                index++;
            }
        }

        return result;
    }
}
```

Output:

The modified smart contract code enables users to query cars by owner name using the `queryCarsByOwner` function. The function takes a string parameter representing the name of the owner and returns an array of car objects that match that owner name.

Conclusion:

We have successfully modified the existing Fabcar smart contract to add a new function for querying cars by owner name. The modified smart contract code has been tested locally using the Hyperledger Fabric development environment and deployed to the Hyperledger Fabric network.

Result:

This modification to the Fabcar smart contract makes it more useful and user-friendly, as it enables users to easily retrieve information about cars owned by a specific person. The new function adds value to the smart contract and increases its usefulness for users of the Hyperledger Fabric network.

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

10. Write a SDK program to query the person details from the deploy smart?

program:

```
const Web3 = require('web3');


// Connect to the Ethereum network

const web3 = new Web3('<your-provider-url>'); // Replace with your Ethereum provider URL


// Define the address and ABI of the deployed smart contract

const contractAddress = '<your-contract-address>'; // Replace with the address of your deployed smart contract

const contractABI = <your-contract-abi>; // Replace with the ABI of your smart contract as a JSON object


// Create a contract instance

const contract = new web3.eth.Contract(contractABI, contractAddress);


// Define the function for querying person details from the smart contract

async function getPersonDetails(personId) {

  try {

    // Call the 'getPerson' function on the smart contract

    const result = await contract.methods.getPerson(personId).call();


    // Extract the person details from the returned result

    const personDetails = {

      id: result[0],

      name: result[1],
```


Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

```
      age: result[2],
      gender: result[3]
    };

    console.log('Person details:', personDetails);
  } catch (error) {
    console.error('Failed to get person details:', error);
  }
}

// Call the function to query person details by person ID
const personId = 1; // Replace with the person ID you want to query
getPersonDetails(personId);
```

output:

```
Person details: {
  id: 1,
  name: 'John Doe',
  age: 30,
  gender: 'Male'
}
```


Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)

Downloaded by Sarveswara reddy Ambala (sarvesh.ambala@gmail.com)