Lab Manual of Blockchain [CSIT-802]

B. Tech. VIII Semester

Jan - June 2024

Department of Computer Science and Information Technology

Submitted to

Prof.Vandana Kate

Designation, CSIT Dept.

Submitted By

Abhidatt Sharma

0827CI201005

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

Department of Computer Science and Information Technology

Certificate

This is to certify that the experimental work entered in this journalas per the B Tech IV year syllabus prescribed by the RGPV was done by Mr. / Ms. Abhidatt Sharma BTech VIII semester CI in the Blockchain Laboratory of this institute during the academic year Jan June 2024

Signature of
Faculty

Experiment 1: Setting Up Metamask

Experiment 2: Multiple Accounts & Balance Transfer

Experiment 3: Setting Up Ganache

Experiment 4: Custom RPC Network with Metamask & Ganache

Experiment 5: Solidity Smart Contract for Balance Transfer

Experiment 6: Solidity Program Demonstrating Exception Handling

---

Solidity Programs

1. **Hello World:**
    - **Description: This is a simple Solidity smart contract that stores a greeting message and allows anyone to retrieve it using the getGreeting() function.**
    - **Features:**
        - **State variable for storing the greeting message.**
        - **Constructor to initialize the greeting message.**
        - **Getter function getGreeting() to retrieve the stored greeting message.**
2. **Simple Voting System:**
    - **Description: This contract implements a basic voting system where each address can vote for a candidate represented by a bytes32 identifier. It prevents double voting by tracking whether an address has voted already and maintains a tally of votes received for each candidate.**
    - **Features:**
        - **Mapping to track whether an address has voted.**
        - **Mapping to maintain the tally of votes received for each candidate.**
        - **Function vote() to allow an address to cast a vote.**
        - **Function getVotesForCandidate() to query the votes received by a specific candidate.**
3. **Simple Storage:**
    - **Description: Stores and retrieves a single uint256 value.**

- **Features:**
    - **State variable to store the uint256 value.**
    - **Setter function to update the stored value.**
    - **Getter function to retrieve the stored value.**

4. **Coin Flip:**
    - **Description: Simulates a coin flip using the current timestamp.**
    - **Features:**
        - **Utilizes the block timestamp to determine the outcome of the coin flip.**
        - **Conditional statement to check if the timestamp is even or odd.**
5. **Simple Payment:**
    - **Description: Allows anyone to send Ether to the contract.**
    - **Features:**
        - **Receive function to accept Ether sent to the contract.**
        - **Accessible balance of the contract, which can be checked using the address balance.**
6. **Split Payment:**
    - **Description: Splits received Ether equally between two recipients.**
    - **Features:**
        - **Use of address payable to specify recipient addresses.**
        - **Transfer function to send Ether to the specified recipients.**
7. **Simple Token:**
    - **Description: Implements a basic token system for transferring tokens between addresses.**
    - **Features:**
        - **Mapping to track token balances of addresses.**
        - **Functionality to transfer tokens between addresses.**
8. **Election:**

- **Description: Enables users to vote for candidates identified by bytes32 values.**
- **Features:**
    - **Mapping to track votes for each candidate.**
    - **Voting mechanism to prevent double voting.**

9. **Simple Auction:**
    - **Description: Implements a simple auction system for placing bids.**
    - **Features:**
        - **Tracking of the highest bidder and highest bid.**
        - **Bid validation to ensure that the bid amount is higher than the current highest bid.**
        - **Transfer function to transfer funds to the highest bidder.**
10. **Token Sale:**
    - **Description: Allows users to buy tokens by sending Ether.**
    - **Features:**
        - **Functionality to purchase tokens by sending Ether.**
        - **Validation to ensure that the correct amount of Ether is sent.**
11. **Simple Lottery:**
    - **Description: Conducts a lottery where participants can enter and a winner is randomly chosen.**
    - **Features:**
        - **Array manipulation to manage participants.**
        - **Random number generation to select a winner.**
12. **Simple Crowdfunding:**
    - **Description: Enables users to contribute funds towards a goal and checks whether the goal has been reached.**
    - **Features:**
        - **Goal tracking to monitor the progress towards the funding goal.**
        - **Accepting contributions from users.**

- **Checking whether the funding goal has been reached.**

13. Simple Escrow:

- **Description: Facilitates a basic escrow service where funds are held until certain conditions are met.**
- **Features:**
    - **Multiple parties involvement.**
    - **Conditional release of funds based on predefined conditions.**

14. Decentralized Exchange (DEX):

- **Description: Implements a decentralized exchange for trading tokens.**
- **Features:**
    - **Order book management.**
    - **Execution of trades between parties.**

15. Multi-Signature Wallet:

- **Description: Creates a multi-signature wallet requiring multiple signatures for transactions.**
- **Features:**
    - **Management of multiple owners and their respective signatures.**
    - **Validation of transactions based on required signatures.**

Experiment 1

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

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 facilitates interaction 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:

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

Experiment 2

Aim: To create mul tiple 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.

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: Metamask simplifies the creation and management of multiple accounts and facilitates fund transfers between them. Transaction details are recorded on the blockchain, and users can verify them using 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. Experimenting with different transaction types and amounts can enhance understanding of the Ethereum network.

Experiment 3

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.

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.

Experiment 4

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 (e.g., 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 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.

```
pragma solidity ^0.8.0;


contract BalanceTransfer {
    address public owner;
    mapping(address => uint256) public balances;


    constructor() {
```

```
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only contract owner can call this function");
        _;
    }

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

    function transfer(address payable _recipient, uint256 _amount) public onlyOwner {
        require(address(this).balance >= _amount, "Insufficient balance in the contract");
        _recipient.transfer(_amount);
        balances[_recipient] += _amount;
    }
}
```

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

Conclusion: By creating a custom RPC network in Metamask and connecting it with the Ganache tool, you can test and develop smart contracts locally and transfer test ether between accounts. This setup enables efficient and secure development and testing of Ethereum-based applications 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

Experiment 5

Title: Solidity Smart Contract for Balance Transfer

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 (e.g., Remix, Visual Studio Code with Solidity extension).
2. Create a new Solidity file.
3. Copy and paste the provided code into the file.
4. Compile the contract using the Solidity compiler to ensure there are no errors or warnings.
5. Deploy the compiled contract to the Ethereum network.
6. To transfer balance from the contract to other accounts, call the transfer function with the recipient address and the amount to transfer as arguments.
7. To check the balance of the contract, call the getContractBalance function.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract BalanceTransfer {

    address public owner;


    constructor() {

        owner = msg.sender;

    }


    // Function to transfer balance from contract to other accounts

    function transfer(address payable _to, uint256 _amount) public payable {
```

```
        require(msg.sender == owner, "Only the contract owner can initiate transfers");
        require(address(this).balance >= _amount, "Insufficient balance in the contract");

        // Transfer funds to the recipient
        _to.transfer(_amount);
    }

    // Function to check the balance of the contract
    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Conclusion: The provided Solidity smart contract facilitates balance transfers from the contract to other Ethereum network accounts. By utilizing the transfer function, the contract owner can initiate transfers securely. Furthermore, monitoring the contract's balance through the getContractBalance function ensures sufficient funds for transfers.

Result: Following this guide enables the creation of a versatile Solidity smart contract capable of executing balance transfers, suitable for various use cases like rewards distribution and token disbursement.

Experiment 6

Title: Solidity Program Demonstrating Exception Handling

Aim: To develop a Solidity program showcasing exception handling.

Requirements:

- Solidity compiler (version 0.8.0 or above)
- Development environment (e.g., Remix, Visual Studio Code with Solidity extension)

Procedure:

1. Open your chosen development environment and create a new Solidity file.
2. Copy and paste the provided code into the file.

3. Compile the contract using the Solidity compiler to identify any errors or warnings.
4. Deploy the contract to the Ethereum network.
5. Test exception handling by invoking the divide, multiply, and transfer functions with various arguments.

```
pragma solidity ^0.8.0;

contract ExceptionHandling {

    function divide(uint256 _numerator, uint256 _denominator) public pure returns (uint256) {

        require(_denominator != 0, "Denominator cannot be zero");

        return _numerator / _denominator;

    }


    function multiply(uint256 _a, uint256 _b) public pure 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);

    }

}
```

Conclusion: Exception handling in Solidity is crucial for ensuring the robustness and security of smart contracts. The **require** statement is utilized to enforce conditions that must be satisfied for functions to execute, while the **revert** statement is employed to revert transactions with

custom error messages. Testing the **divide**, **multiply**, and **transfer** functions with various arguments helps verify the effectiveness of exception handling in preventing unexpected behavior.

Result: Following this guide enables you to develop Solidity programs that incorporate exception handling mechanisms, enhancing the reliability and safety of your smart contracts.

Solidity Programs:

1. **Hello World**:

   Description: This is a simple Solidity smart contract that stores a greeting message and allows anyone to retrieve it using the **getGreeting()** function.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract HelloWorld {

    string public greeting;


    constructor() {

        greeting = "Hello, World!";

    }


    function getGreeting() public view returns (string memory) {

        return greeting;

    }

}
```

2. **Simple Voting System**:

   This contract implements a basic voting system where each address can vote for a candidate represented by a **bytes32** identifier. It prevents double voting by tracking whether an address has voted already and maintains a tally of votes received for each candidate. The **vote** function allows an address to cast a vote, and **getVotesForCandidate** function allows querying the votes received by a specific candidate.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract SimpleVoting {

    mapping(address => bool) public hasVoted;

    mapping(bytes32 => uint256) public votesReceived;


    function vote(bytes32 candidate) public {
```

```
        require(!hasVoted[msg.sender], "You have already voted.");
        votesReceived[candidate]++;
        hasVoted[msg.sender] = true;
    }

    function getVotesForCandidate(bytes32 candidate) public view returns (uint256) {
        return votesReceived[candidate];
    }
}
```

3. **Simple Storage**:

   This contract stores a single uint256 value and provides functions to set and retrieve it.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 public data;

    function setData(uint256 _data) public {
        data = _data;
    }

    function getData() public view returns (uint256) {
        return data;
    }
}
```

4. **Coin Flip**:

   This contract simulates a coin flip by returning true if the current timestamp is even, false otherwise.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract CoinFlip {

    function flipCoin() public view returns (bool) {

        return block.timestamp % 2 == 0;

    }

}
```

5. **Simple Payment**:

   This contract allows anyone to send Ether to it and provides a function to check its balance.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract SimplePayment {

    receive() external payable {}


    function getBalance() public view returns (uint256) {

        return address(this).balance;

    }

}
```

6. **Split Payment**:

This contract splits the received Ether equally between two specified recipients.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;
```

```
contract SplitPayment {
    function split(address payable recipient1, address payable recipient2) public payable {
        require(msg.value > 0, "No value sent");
        uint256 amount = msg.value / 2;
        recipient1.transfer(amount);
        recipient2.transfer(amount);
    }
}
```

7. **Simple Token**:

   This contract implements a simple token system where users can transfer tokens to each other.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleToken {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

8. **Election**:

   This contract allows users to vote for candidates identified by bytes32 values.

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.0;

contract Election {
    mapping(bytes32 => uint256) public votes;

    function vote(bytes32 candidate) public {
        votes[candidate]++;
    }
}
```

9. **Simple Auction**:

This contract implements a simple auction system where users can place bids.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleAuction {
    address public highestBidder;
    uint256 public highestBid;

    function bid() public payable {
        require(msg.value > highestBid, "Bid too low");
        if (highestBid != 0) {
            payable(highestBidder).transfer(highestBid);
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

10. **Token Sale**:

    This contract allows users to buy tokens by sending Ether.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TokenSale {
    mapping(address => uint256) public balances;

    function buyTokens(uint256 amount) public payable {
        require(msg.value == amount, "Incorrect payment amount");
        balances[msg.sender] += amount;
    }
}
```

11. **Simple Lottery**:

    This contract implements a simple lottery system where users can enter and a winner is randomly chosen.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleLottery {
    address[] public participants;

    function enter() public {
        participants.push(msg.sender);
    }

    function pickWinner() public {
```

```
        require(participants.length > 0, "No participants");
        uint256 index = random() % participants.length;
        payable(participants[index]).transfer(address(this).balance);
        delete participants;
    }

    function random() private view returns (uint256) {
        return uint256(keccak256(abi.encodePacked(block.difficulty, block.timestamp, participants)));
    }
}
```

12. **Simple Crowdfunding**:

    This contract allows users to contribute funds towards a goal and checks whether the goal has been reached.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleLottery {
    address[] public participants;

    function enter() public {
        participants.push(msg.sender);
    }

    function pickWinner() public {
        require(participants.length > 0, "No participants");
        uint256 index = random() % participants.length;
        payable(participants[index]).transfer(address(this).balance);
```

```
        delete participants;
    }

    function random() private view returns (uint256) {
        return uint256(keccak256(abi.encodePacked(block.difficulty, block.timestamp, participants)));
    }
}
```

13. Simple Escrow

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleEscrow {
    address public payer;
    address public payee;
    address public arbiter;
    uint256 public amount;

    constructor(address _payer, address _payee, address _arbiter, uint256 _amount) {
        payer = _payer;
        payee = _payee;
        arbiter = _arbiter;
        amount = _amount;
    }

    function releaseToPayee() public {
        require(msg.sender == arbiter, "Only arbiter can release funds");
        payable(payee).transfer(amount);
    }

    function releaseToPayer() public {
        require(msg.sender == arbiter, "Only arbiter can release funds");
        payable(payer).transfer(amount);
    }

    function cancel() public {
        require(msg.sender == arbiter, "Only arbiter can cancel");
        payable(payer).transfer(amount);
```

```
    }
}
```

14. **Decentralized Exchange (DEX)**:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DecentralizedExchange {
    mapping(address => mapping(address => uint256)) public tokens;

    function deposit(address token, uint256 amount) public {
        tokens[token][msg.sender] += amount;
    }

    function withdraw(address token, uint256 amount) public {
        require(tokens[token][msg.sender] >= amount, "Insufficient balance");
        tokens[token][msg.sender] -= amount;
    }

    function trade(address token, uint256 amount, address recipient) public {
        require(tokens[token][msg.sender] >= amount, "Insufficient balance");
        tokens[token][msg.sender] -= amount;
        tokens[token][recipient] += amount;
    }
}
```

15. **Multi-Signature Wallet**:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
contract MultiSigWallet {
    address[] public owners;
    mapping(address => bool) public isOwner;
    uint256 public numConfirmationsRequired;

    constructor(address[] memory _owners, uint256 _numConfirmationsRequired) {
        require(_owners.length > 0, "Owners required");
        require(_numConfirmationsRequired > 0 && _numConfirmationsRequired <= _owners.length, "Invalid number of confirmations");

        for (uint256 i = 0; i < _owners.length; i++) {
            address owner = _owners[i];
            require(owner != address(0), "Invalid owner");
            require(!isOwner[owner], "Owner not unique");
            isOwner[owner] = true;
            owners.push(owner);
        }

        numConfirmationsRequired = _numConfirmationsRequired;
    }

    function submitTransaction(address destination, uint256 value, bytes memory data) public {
        // Submit transaction logic
    }

    function confirmTransaction(uint256 transactionId) public {
```

```
        // Confirm transaction logic
    }

    function revokeConfirmation(uint256 transactionId) public {
        // Revoke confirmation logic
    }

    function executeTransaction(uint256 transactionId) public {
        // Execute transaction logic
    }
}
```