School: ..... Campus: .....

Academic Year: ..... Subject Name: ..... Subject Code: .....

Semester: ..... Program: ..... Branch: ..... Specialization: .....

Date: .....

Applied and Action Learning

(Learning by Doing and Discovery)

Name of the Experiement : 2 Read the Chain – Web3.js Basics

* Coding Phase: Pseudo Code / Flow Chart / Algorithm

Introduction

- **Blockchain Data Access** – Web3.js allows developers to read data stored on the blockchain (Ethereum, BSC, Polygon, etc.) such as account balances, transaction details, and block info.
- **No Gas Fees for Reading** – Reading (calling eth_call) is free because it doesn't change the blockchain state, unlike transactions (eth_sendTransaction).
- **Functions for Reading** – Common Web3.js methods include web3.eth.getBalance(), web3.eth.getBlock(), web3.eth.getTransaction(), and contract read functions using myContract.methods.methodName().call().
- **Smart Contract Interaction** – You can connect to deployed smart contracts using their **ABI** and address, then read variables and return values from functions without modifying the state.
- **Use Cases** – Reading is used for showing token balances, NFT metadata, transaction history, block details, and DApp dashboards.

Algorithm:

1. Open Remix IDE and write the SimpleStorage.sol smart contract.
2. Compile the smart contract using the Solidity compiler in Remix.
3. Copy the generated ABI after successful compilation.
4. Deploy the contract to the Sepolia Testnet using MetaMask.
5. Copy the deployed contract address.
6. Create a React frontend project using create-react-app.
7. Add the contract address and network information to the .env file.
8. Install web3.js to interact with the blockchain.
9. Use the ABI and contract address to connect the frontend with the smart contract.
10. Design the UI in App.js using ethers.js to store and retrieve data.

* Softwares used

1. MetaMask Wallet
2. Remix IDE
3. Brave browser

*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Testing Phase: Compilation of Code (error detection)

Go to remix ide and write a smart contract on simplestorage.sol and compile it

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Counter {
    uint public count;
    constructor(uint _start) {    infinite gas 132000 gas
        count = _start;
    }
    function increment() public {    infinite gas
        count += 1;
    }
    function decrement() public {    infinite gas
        require(count > 0, "Counter is already at zero");
        count -= 1;
    }
    function getCount() public view returns (uint) {    2453 gas
        return count;
    }
}
```

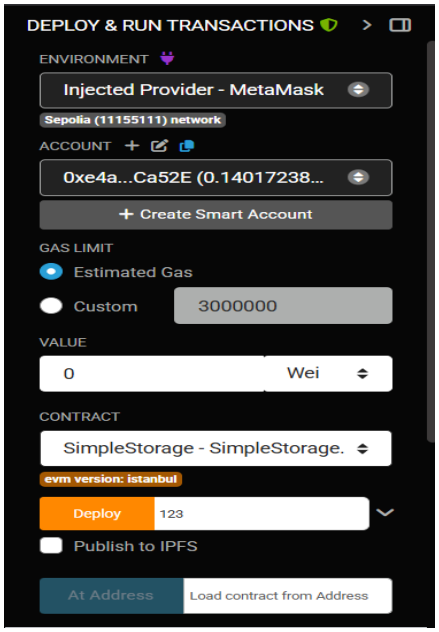
After compile the smart contract there is a ABI of the smart contract

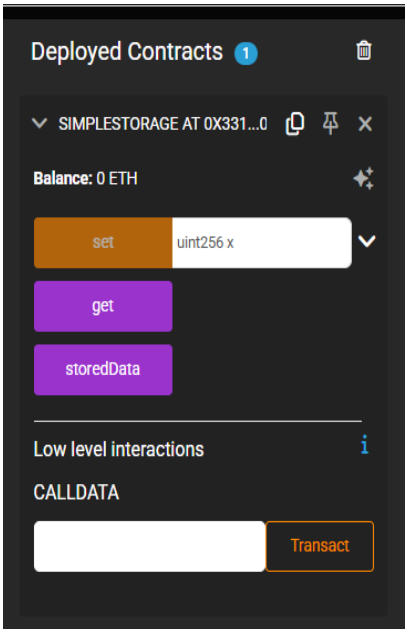
```
[
    {
        "inputs": [
            {
                "internalType": "uint256",
                "name": "_start",
                "type": "uint256"
            }
        ],
        "stateMutability": "nonpayable",
        "type": "constructor"
    },
    {
        "inputs": [],
        "name": "count",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "decrement",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "getCount",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "increment",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```


*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Testing Phase: Compilation of Code (error detection)

After compilation ,deploy the smart contract and choose the enviornment as injector provider-metamask then give some value and start deploy





In this Smart contract we have two accessible libraries one is ether.js and another is web3.js we have to work on Ethers.js

*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Implementation Phase: Final Output (no error)

Now we have to work on frontend first create a folder for your frontend then open terminal to install the react modules . Then create a ABI.js file inside your src folder where we have to store the abi of our smart contract and then create a .env file in the root of the project folder to dtore contract address and tectnet network

simple-storage-dapp > src > .env

```
REACT_APP_CONTRACT_ADDRESS=0xa62463A56EE9D742F810920F56cEbc4B696eBd0a
REACT_APP_NETWORK=sepolia
```

Now in App.js write your frontend code and wallet connection code importing web3.

simple-storage-dapp > src > App.js > App

```
import React, { useEffect, useState } from "react";
import Web3 from "web3";
import { CONTRACT_ABI } from "./abi";

// 🔗 Replace with your deployed contract address from Remix
const CONTRACT_ADDRESS = "0xa62463A56EE9D742F810920F56cEbc4B696eBd0a";

function App() {
  const [web3, setWeb3] = useState(null);
  const [account, setAccount] = useState("");
  const [contract, setContract] = useState(null);
  const [inputValue, setInputValue] = useState("");
  const [storedValue, setStoredValue] = useState("");
  const [loading, setLoading] = useState(false);
  const [status, setStatus] = useState("");

  // 🔄 Connect Wallet
  const connectWallet = async () => {
    if (window.ethereum) {
      try {
        const web3Instance = new Web3(window.ethereum);
        setWeb3(web3Instance);

        const accounts = await window.ethereum.request({
          method: "eth_requestAccounts",
        });
        setAccount(accounts[0]);

        const contractInstance = new web3Instance.eth.Contract(
          CONTRACT_ABI,
          CONTRACT_ADDRESS
        );
        setContract(contractInstance);

        const currentValue = await contractInstance.methods.get().call();
        setStoredValue(currentValue);
        setStatus("✅ Wallet connected");
      } catch (error) {
        console.error("Wallet connection error:", error);
        setStatus("❌ Failed to connect wallet.");
      }
    } else {
      alert("Please install MetaMask to use this DApp.");
    }
  };
```

```
  // ⚪ Handle Submit
  const handleSubmit = async () => {
    if (!inputValue) return alert("Please enter a number.");
    if (!contract || !account) return alert("Wallet not connected.");

    try {
      setLoading(true);
      setStatus("⏳ Sending transaction...");

      await contract.methods.set(inputValue).send({ from: account });

      const updatedValue = await contract.methods.get().call();
      setStoredValue(updatedValue);
      setInputValue("");
      setStatus("✅ Number updated on blockchain!");
    } catch (error) {
      console.error(error);
      setStatus("❌ Transaction failed.");
    } finally {
      setLoading(false);
    }
  };

  return (
    <div style={{ padding: "40px", fontFamily: "Arial", maxWidth: "600px", margin: "auto" }}>
      <h2>🔐 Simple Storage DApp</h2>

      {!account ? (
        <button
          onClick={connectWallet}
          style={{
            padding: "10px 20px",
            backgroundColor: "🟦#007bff",
            color: "white",
            border: "none",
            borderRadius: "5px",
            marginBottom: "20px",
          }}
        >
```


*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Implementation Phase: Final Output (no error)

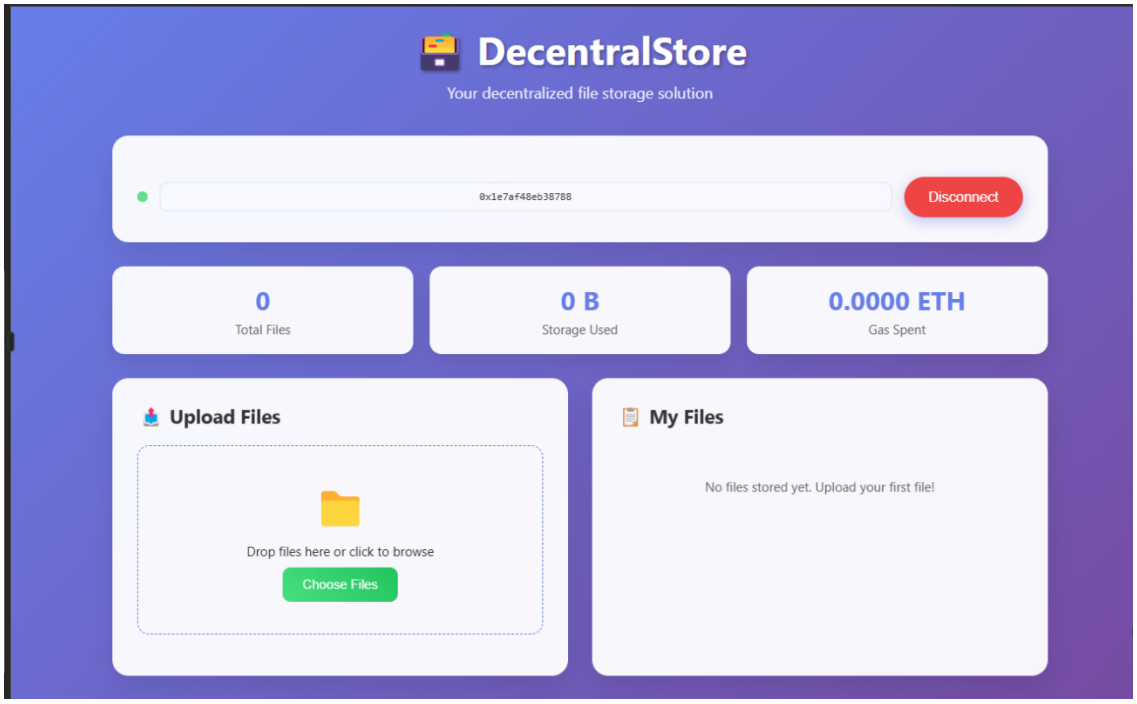
```
            🔌 Connect Wallet
          </button>
        ) : (
          <div style={{ marginBottom: "20px" }}>
            <p><strong>🟣 Wallet:</strong> {account}</p>
            <p><strong>📦 Stored Number:</strong> {storedValue}</p>
          </div>
        )}

        {account && (
          <>
            <div>
              <input
                type="number"
                value={inputValue}
                onChange={(e) => setInputValue(e.target.value)}
                placeholder="Enter a number"
                style={{ padding: "8px", width: "70%", marginRight: "5px" }}
                disabled={loading}
              />
              <button onClick={handleSubmit} disabled={loading}>
                {loading ? "⏳ Updating..." : "Set Number"}
              </button>
            </div>

            {status && (
              <p style={{ marginTop: "20px", color: status.includes("✅") ? "green" : "red" }}>
                {status}
              </p>
            )}
          </>
        )}
      </div>
    );
  }

  export default App;
```

After write all the coder now for frontend design write the css .after writing all coder now install the web3 packages inside your frontend folder to install all the packages of web3 the command is -npm install web3
after installing all the packages now to run the frontend write the command
npm start

```
> vite


  VITE v7.0.6  ready in 1510 ms

  → Local:   http://localhost:5173/

  VITE v7.0.6  ready in 1510 ms
```


*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Implementation Phase: Final Output (no error)



Now in this frontend you can update value and check stored value

Observations

1.Ethers.js provides a lightweight and modular approach for interacting with Ethereum smart contract.
2. It simplifies wallet connection and contract function calls using a clean and modern syntax.
3.The library ensures better security and improved developer experience compared to older Web3.js practices.

ASSESSMENT

Rubrics	Full Mark	Marks Obtained	Remarks
Concept	10		
Planning and Execution/ Practical Simulation/ Programming	10		
Result and Interpretation	10		
Record of Applied and Action Learning	10		
Viva	10		
Total	**50**		

Signature of the Student:

Name :

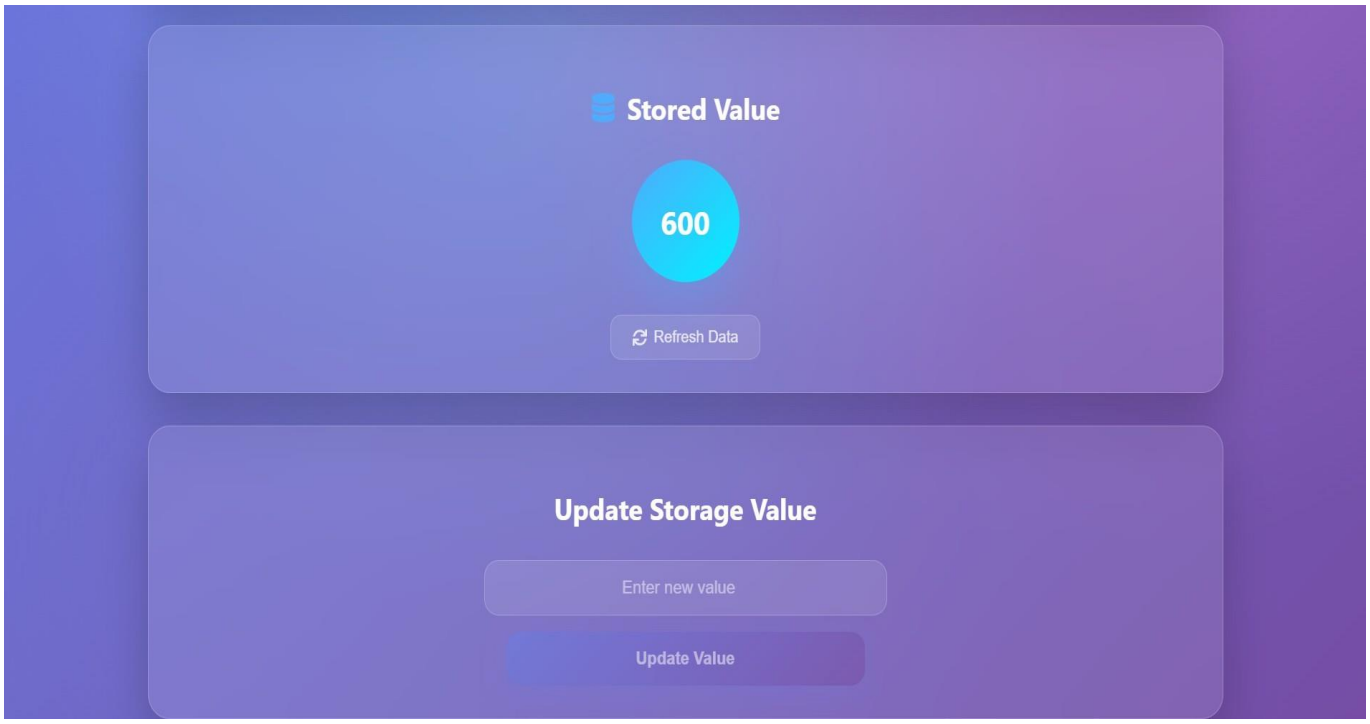
Signature of the Faculty:

Regn. No. :

Page No.....

***As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.***

* Implementation Phase: Final Output (no error)



Now in this frontend you can update value and check stored value

* Observations

1. Ethers.js provides a lightweight and modular approach for interacting with Ethereum smart contracts.
2. It simplifies wallet connection and contract function calls using a clean and modern syntax.
3. The library ensures better security and improved developer experience compared to older Web3.js practices.

ASSESSMENT

Rubrics	Full Mark	Marks Obtained	Remarks
Concept	10		
Planning and Execution/ Practical Simulation/ Programming	10		
Result and Interpretation	10		
Record of Applied and Action Learning	10		
Viva	10		
Total	**50**		

Signature of the Student:

Name :

Signature of the Faculty:

Regn. No. :

**As applicable according to the experiment.*
Two sheets per experiment (10-20) to be used.