Assignment No: 03

Title: Build and deploy a smart contract with testnet.

Objective:

- To learn how to deploy a smart contract on a test network.
- To understand how to connect a web application with an Ethereum wallet.
- To interact with a deployed smart contract through a connected wallet.

Problem Statement:

Connect to any Ethereum wallet. e.g. MetaMask. Deploy the contract with testnet. Connect wallet with your webapp. Call the deployed contract through your web app. Then store the wave messages from users in arrays using structs.

Theory / Procedure / Diagrams:

Ethereum smart contracts are self-executing programs stored on the blockchain. They allow peer-to-peer interaction without intermediaries. In this experiment, we deploy a simple "Wave" contract that stores user messages (waves) on-chain. MetaMask acts as a bridge between the browser and the blockchain, while Remix IDE is used for contract development and deployment.

1. MetaMask:

MetaMask is a self-custody wallet that allows users to manage accounts, store private keys, and interact with decentralized applications directly in their browser. It supports multiple networks, including Ethereum mainnet and testnets such as Goerli and Sepolia.

2. Remix IDE:

Remix is a browser-based Solidity development environment. It allows users to write, compile, and deploy smart contracts easily without any local setup. It integrates directly with MetaMask via the "Injected Provider – MetaMask" option.

3. Smart Contract Logic:

The contract stores user messages in an array of structs. Each struct contains the sender's address, message text, and timestamp.

```
struct Wave {
    address sender;
    string message;
    uint256 timestamp;
}
Wave[] public waves;
```

Procedure:

Step 1: Open Remix IDE

Visit the official Remix IDE website at https://remix.ethereum.org

Remix provides an in-browser environment for writing, compiling, and deploying smart contracts without installing any local dependencies.

Step 2: Create a new contract file

In the File Explorer panel, navigate to the contracts directory.

Create a new Solidity file named WavePortal.sol.

Write the smart contract code inside it including a Wave struct, an array to store messages, and functions to send and retrieve waves.

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


contract WavePortal {
    struct Wave {
        address waver; // The address of the user who waved
        string message; // The message the user sent
        uint256 timestamp; // The timestamp when the user waved
    }


    Wave[] public waves; // Array to store all waves
```

```
    uint256 public totalWaves; // Counter for total waves

    event NewWave(address indexed from, string message, uint256 timestamp);

    constructor() {
        totalWaves = 0;
    }

    function wave(string memory _message) public {
        waves.push(Wave(msg.sender, _message, block.timestamp));
        totalWaves += 1;
        emit NewWave(msg.sender, _message, block.timestamp);
    }

    function getAllWaves() public view returns (Wave[] memory) {
        return waves;
    }

    function getTotalWaves() public view returns (uint256) {
        return totalWaves;
    }
}
```

Step 3: Compile the Contract

Go to the Solidity Compiler tab on the left sidebar.

Select the appropriate compiler version (e.g., 0.8.20).

Click the Compile WavePortal.sol button.

Ensure the compilation completes successfully with no errors.

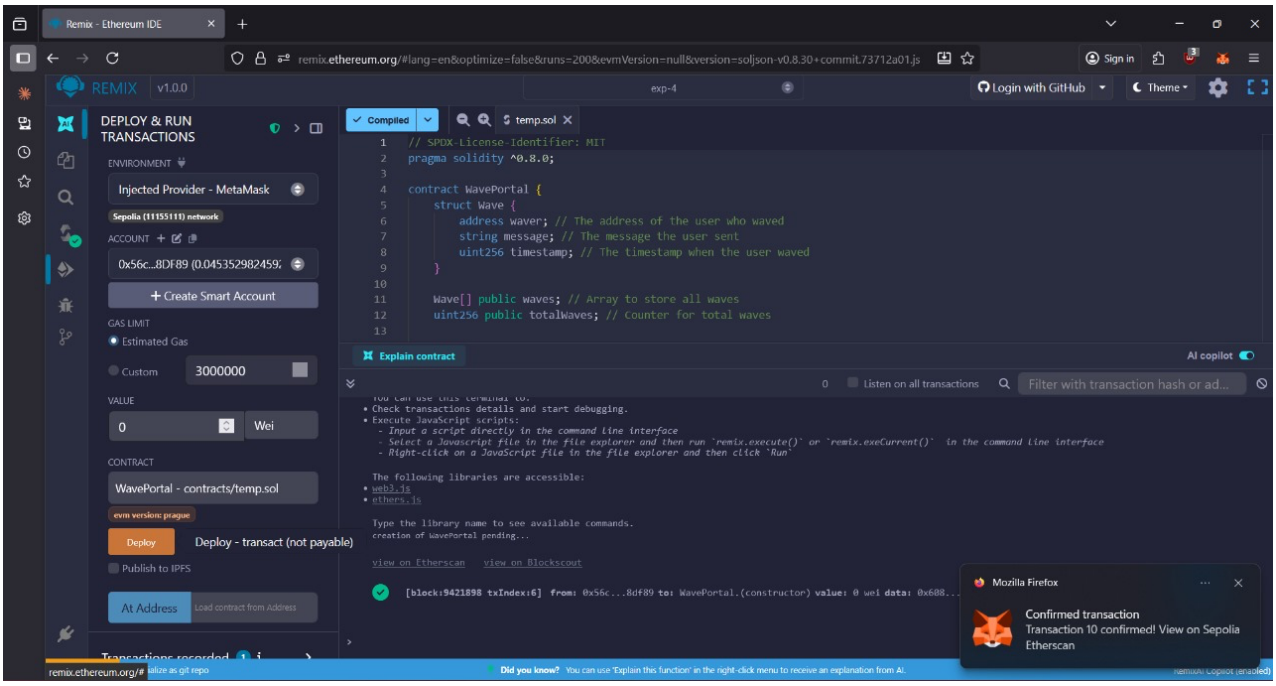
Step 4: Connect MetaMask Wallet

Open the Deploy & Run Transactions tab.

In the Environment dropdown, select Injected Provider – MetaMask.

A MetaMask popup will appear asking for connection permission.

Approve the connection and ensure MetaMask is set to the desired test network (e.g., Sepolia).



Step 5: Deploy the Contract

In Remix, click the Deploy button under the selected contract.

MetaMask will open a confirmation window showing the estimated gas fee and deployment details.

Click Confirm to deploy the contract.

Wait for the transaction to complete — once confirmed, the deployed contract address will appear in Remix under the "Deployed Contracts" section.

Step 6: Interact with the Deployed Contract

Expand the WavePortal contract interface under "Deployed Contracts."

Enter a custom message in the input box beside the wave function.

Click transact to send a wave message.

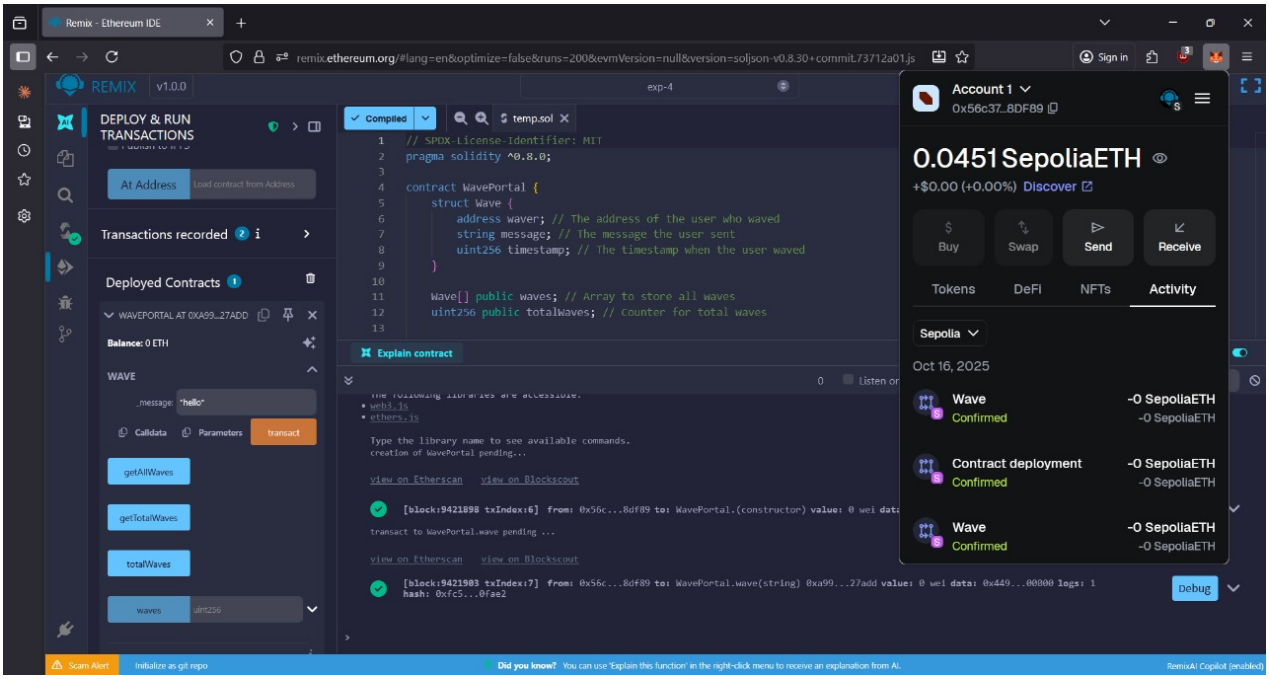
MetaMask will again prompt a transaction confirmation and approve it.

Step 7: Transaction Confirmation and Verification:

After confirmation, Remix will display a success message and transaction hash.

You can click the transaction link to open it on Blockscout (or Etherscan equivalent for the testnet).

The transaction details — including sender address, gas used, and block number — will be visible on the blockchain explorer.

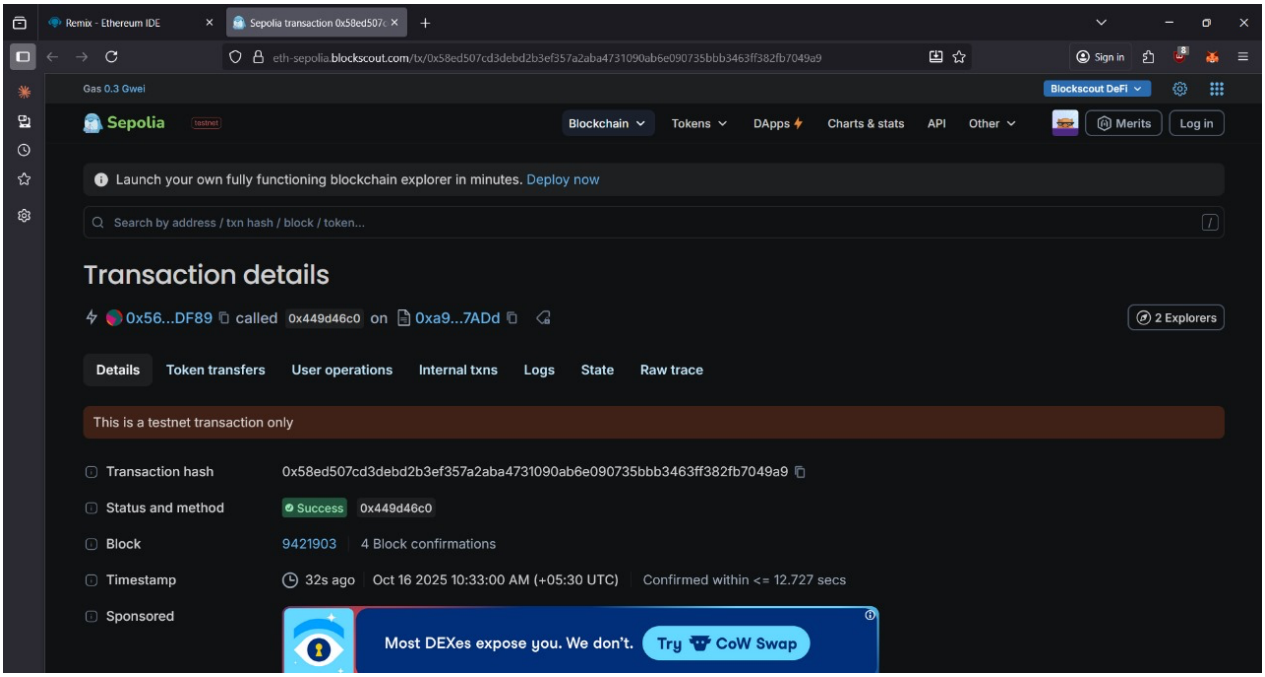


Step 8: View Stored Messages (Optional):

Use the getAllWaves function (if implemented) to retrieve all stored messages.

The list of wave messages, along with sender addresses and timestamps, will be displayed in Remix.

Output:


Remix - Ethereum IDE
Sepolia transaction 0x58ed507c
eth-sepolia.blockscout.com/tx/0x58ed507cd3debd2b3ef357a2aba4731090ab6e090735bbb3463ff382fb7049a9
Sign in
Gas 0.3 Gwei
Blockscout DeFi
Sepolia
testnet
Blockchain
Tokens
DApps
Charts & stats
API
Other
Merits
Log in
Launch your own fully functioning blockchain explorer in minutes. Deploy now
Search by address / txn hash / block / token...
Transaction details
0x56...DF89 called 0x449d46c0 on 0xa9...7ADd
2 Explorers
Details
Token transfers
User operations
Internal txns
Logs
State
Raw trace
This is a testnet transaction only
Transaction hash
0x58ed507cd3debd2b3ef357a2aba4731090ab6e090735bbb3463ff382fb7049a9
Status and method
Success
0x449d46c0
Block
9421903
4 Block confirmations
Timestamp
32s ago
Oct 16 2025 10:33:00 AM (+05:30 UTC)
Confirmed within <= 12.727 secs
Sponsored
Most DEXes expose you. We don't.
Try CoW Swap

Assignment No: 04

Title: Setup Ethereum Development Environment and Create Transactions

Objective:
- To set up a local Ethereum development environment using Hardhat.
- To generate blockchain addresses and simulate transactions on a local network.
- To configure MetaMask for interacting with the local blockchain.

Problem Statement:
Initialize a local Ethereum blockchain using Hardhat, generate test accounts, and perform transactions between these accounts. Configure MetaMask to connect to the local network and
verify transactions.

Theory / Procedure / Diagrams:

Ethereum Development Environment:
Ethereum development requires a simulated blockchain environment for testing and deploying decentralized applications (dApps). Hardhat is a flexible development framework that provides:

- A local Ethereum network (**Hardhat Network**) for testing.
- Built-in tools for compiling, deploying, and debugging smart contracts.
- Pre-configured test accounts with 10,000 ETH each for experimentation.

Generating Addresses & Transactions:

- **Addresses**: Represent user accounts on the blockchain. Each address has a public key (for receiving funds) and a private key (for signing transactions).
- **Transactions**: Actions initiated by accounts (e.g., transferring ETH). Transactions must be signed with the sender's private key and validated by miners.

MetaMask Integration:

MetaMask is a browser-based wallet that:

- Stores private keys securely.
- Connects to blockchain networks (e.g., local Hardhat Network).
- Enables users to view accounts, balances, and transaction history.

Procedure :

Step 1: Initialize Project

mkdir eth-dev-env && cd eth-dev-env

npm init -y # Creates package.json

Step 2: Install Hardhat

npm install --save-dev hardhat@2.22.15

Step 3: Initialize Hardhat Project

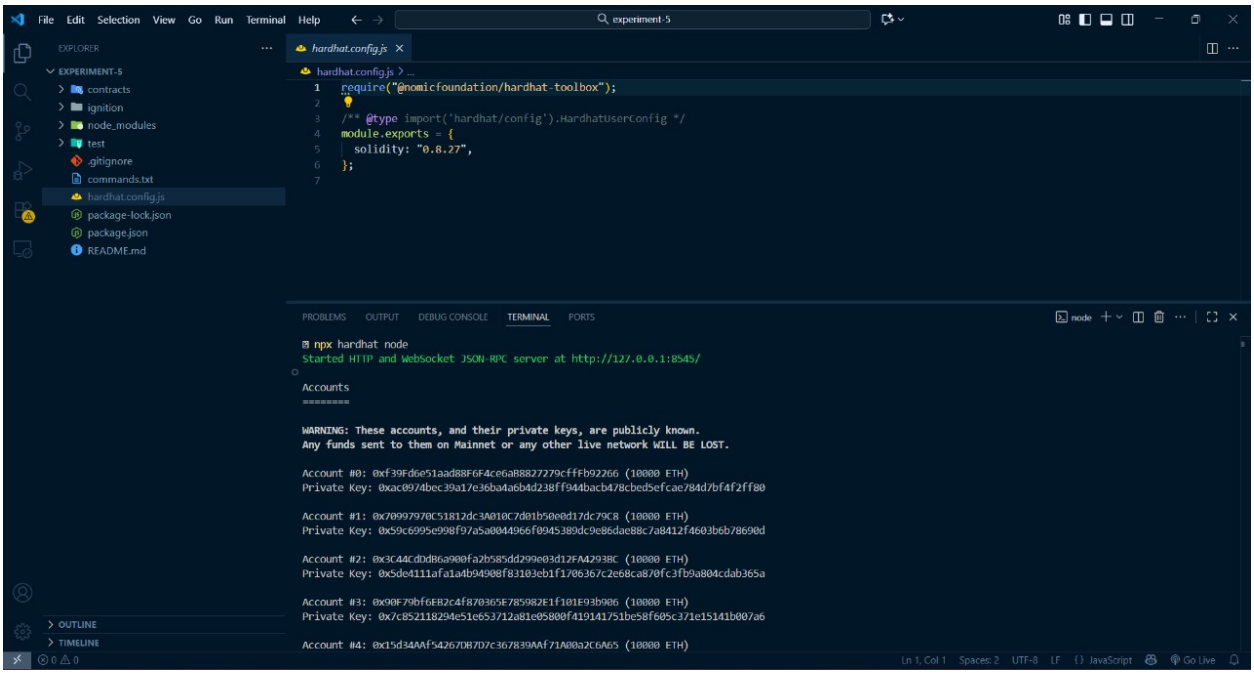
npx hardhat --init

- Select **Create a JavaScript project**.
- Accept defaults for project root, **.gitignore**, and **Install sample project**. *Output:*
- **hardhat.config.js**: Configuration file.
- **contracts/**, **scripts/**, **test/**: Folders for smart contracts, deployment scripts, and tests.

Step 4: Start Local Ethereum Node

npx hardhat node

- 20 test accounts with 10,000 ETH each are generated.
- RPC server runs at **http://127.0.0.1:8545**.



Step 5: Configure MetaMask

1. **Add Hardhat Network to MetaMask**:
    a. Open MetaMask → Network → "Add Network".
    b. Enter:
        i. **Network Name**: **Hardhat Local**
        ii. **RPC URL**: **http://127.0.0.1:8545**
        iii. **Chain ID**: **31337** (default for Hardhat).
        iv. **Currency Symbol**: **ETH**
    c. Click "Save".
2. Import Test Accounts:
    - Copy a private key from npx hardhat node output (e.g., Account 0's private key).
    - In MetaMask: Account → "Import Account" → Paste private key.
    - Result: Account appears with 10,000 ETH balance.

Step 6: Create a Transaction

1. In MetaMask, select the imported account.
2. Click "Send" → Enter recipient address (e.g., Account 1: **0x70997970C51812dc3A010C7d01b50e0d17dc79C8**).
3. Amount: **1 ETH** → Confirm transaction.
4. *Output*:

- Transaction appears in MetaMask history instantly (mined by Hardhat).

Input :

npm init -y

npm install --save-dev hardhat@2.22.15

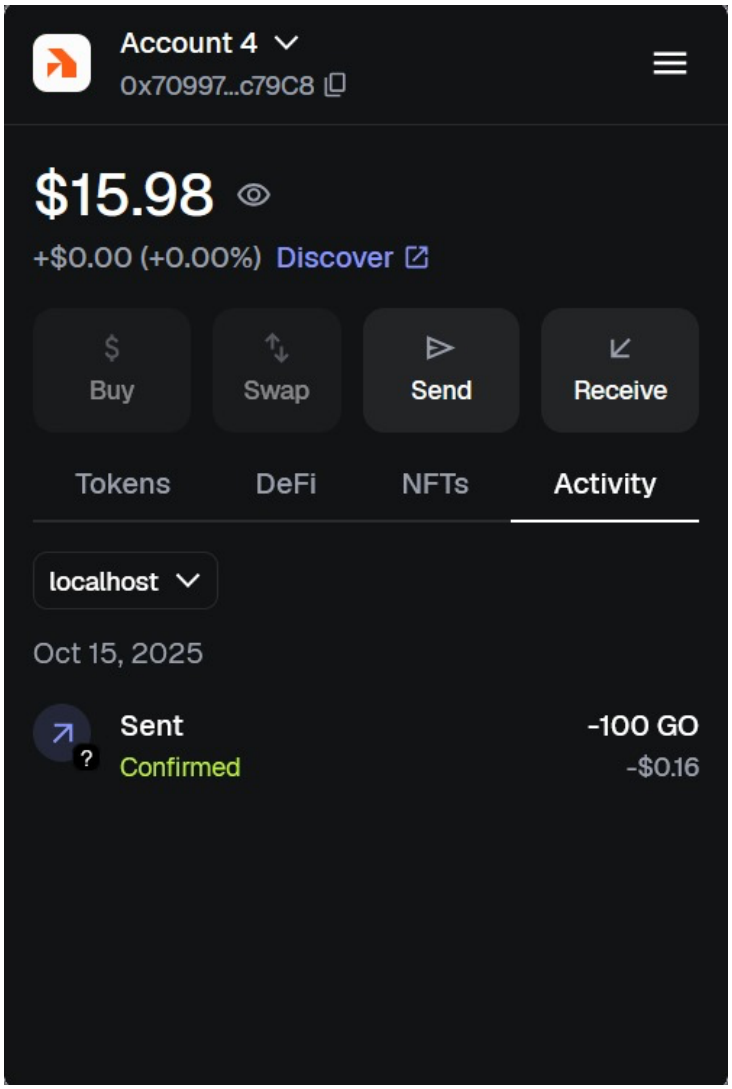
npx hardhat init

npx hardhat node

- **MetaMask Configuration**:
    - RPC URL: **http://127.0.0.1:8545**
    - Chain ID: **31337**
    - Test account private keys (from **npx hardhat node** output).

MetaMask Visualization:

- Account 0 balance: **9999 ETH** (after sending 1 ETH).
- Transaction history shows:
  - **To**: **0x70997970C51812dc3A010C7d01b50e0d17dc79C8**
  - **Amount**: **1 ETH**
  - **Status**: **Success**



Conclusion

Thus, we successfully set up a local Ethereum development environment using Hardhat, generated test accounts, and simulated transactions. MetaMask was configured to interact with the local blockchain, enabling real-time verification of account balances and transactions. This foundation is critical for developing and testing decentralized applications (dApps) before deploying to public testnets or mainnet.

Assignment No: 05

Title: Develop a Hyperledger Fabric Client Application

Objective:

- To design and implement a full-stack web application that interfaces with a Hyperledger Fabric blockchain network.
- To enable blockchain identity management (admin enrollment, user registration) and asset operations (querying/creating assets).
- To develop a real-time wallet status dashboard for monitoring blockchain interactions..

Problem Statement:

Create a client application that:

1. Establishes connectivity to a Hyperledger Fabric blockchain network.
2. Manages blockchain identities:
    - Enrolls an admin user via the Certificate Authority (CA).
    - Registers application users using admin credentials.
3. Facilitates asset operations:
    - Enrolls an admin user via the Certificate Authority (CA).
    - Registers application users using admin credentials.

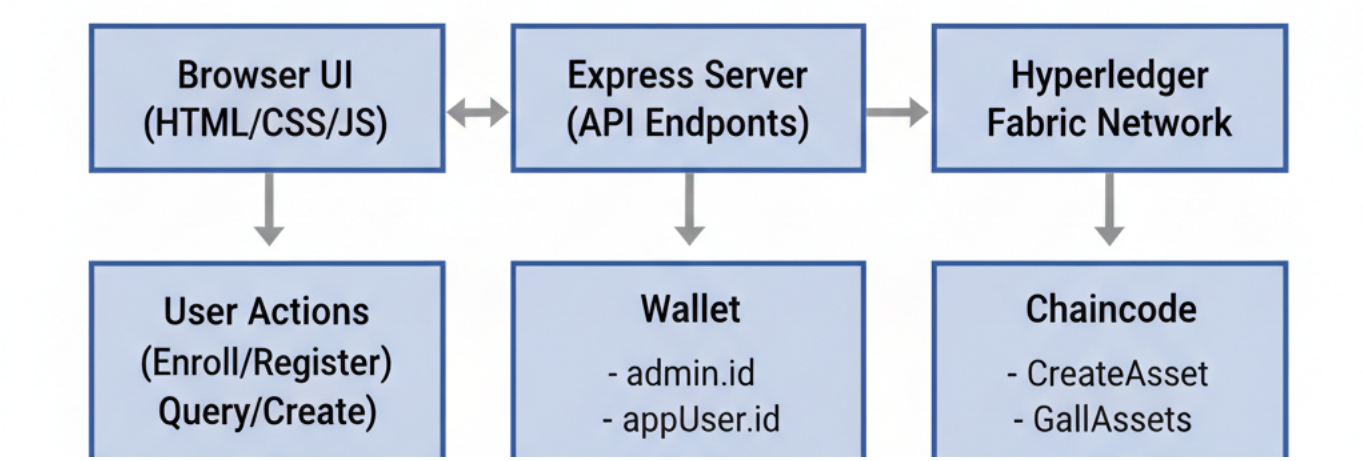
Theory / Procedure / Diagrams:

Theory:

- **Hyperledger Fabric**: A permissioned blockchain framework tailored for enterprise solutions. It utilizes *chaincodes* (smart contracts) to manage assets and transactions.
- **Client Architecture**:
    - **Backend**: Node.js/Express server integrated with the Fabric SDK for blockchain communication.
    - **Frontend**: HTML/CSS/JavaScript interface for user interactions.
    - **Wallet**: Securely stores user identities (admin/appUser) for authentication.
- **Core Components**:
    - **Certificate Authority (CA)**: Issues digital certificates for identity verification.
    - **Peers**: Host chaincodes and maintain the distributed ledger.
    - **Orderers**: Validate and order transactions to ensure consensus

Procedure:

1. **Blockchain Network Setup:**
   - Initialize a Hyperledger Fabric test network with a dedicated channel and Certificate Authority.
   - Deploy the basic chaincode to handle asset operations.
2. **Application Configuration:**
   - Integrate the Fabric network connection profile (connection-org1.json) into the client.
   - Install dependencies (Express, Fabric SDK) and configure the wallet directory.
3. **Backend Implementation:**
   - Develop API endpoints for:
     - Admin enrollment (via CA credentials).
     - User registration (using admin privileges).
     - Asset querying (retrieving ledger data).
     - Asset creation (submitting transactions to the chaincode).
4. **Frontend Development:**
   - Build a responsive UI with:
     - Wallet status indicators (admin/user enrollment state).
     - Forms for asset creation and operation buttons.
     - Real-time feedback for transaction success/errors.
5. **Execution:**
   - Launch the application server and access the web interface.
   - Execute workflows: enroll admin → register user → query assets → create new asset.



WorkFlow

Input (Test Cases / Data Sets):

1) Connection Profile: connection-org1.json (defines network topology and endpoints).

2)Asset Creation Data:

```
{
 "id": "asset10",
 "color": "green",
 "size": 20,
 "owner": "Charlie",
 "appraisedValue": "1000"
}
```

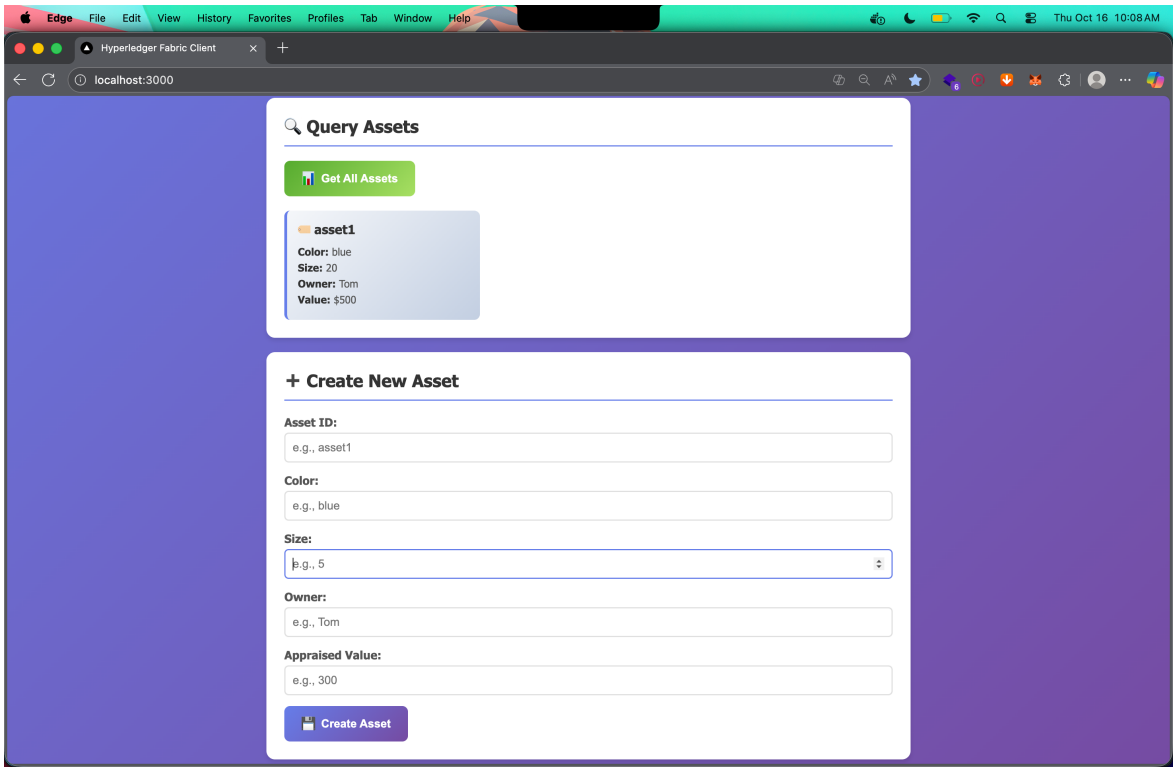
3)Admin Credentials: Default CA credentials (admin/adminpw).

Output (Results / Visualization):

1. **Wallet Status Dashboard:**
   - Admin: Enrolled
   - App User: Registered

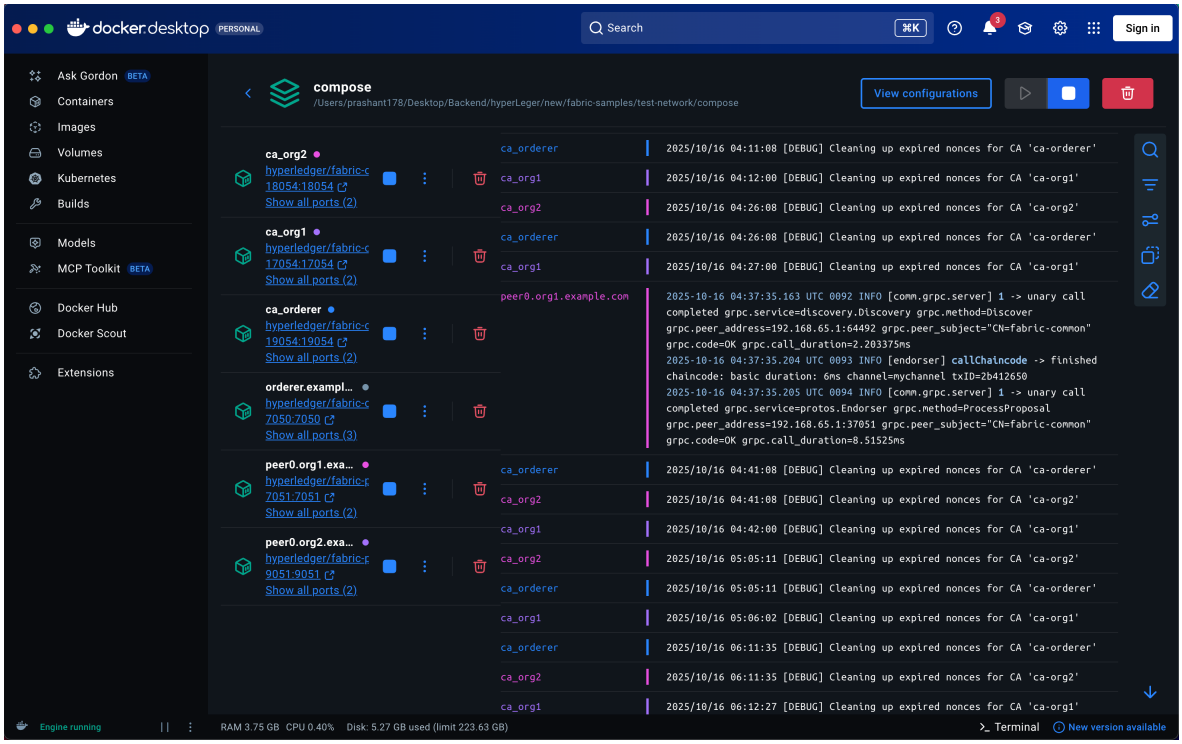
2. ***Asset Query Results:***


```
[
  {
    "ID": "asset1",
    "Color": "blue",
    "Size": 5,
    "Owner": "Tom",
    "AppraisedValue": 300
  },
  {
    "ID": "asset10",
    "Color": "green",
    "Size": 20,
    "Owner": "Charlie",
    "AppraisedValue": 1000
  }
]
```

3. **Transaction Feedback:**
   - Success notifications (e.g., "Asset created successfully").
   - Error alerts for invalid operations (e.g., "Asset ID already exists").

curl -sSL https://bit.ly/2ysbOFE | bash -s
cd fabric-samples/test-network



Conclusion:
This assignment demonstrates the end-to-end development of a Hyperledger Fabric client application, emphasizing:

- **Integration**: Seamless connectivity between the web interface, backend server, and blockchain network.
- **Functionality**: Successful implementation of identity management and asset operations with real-time feedback.
- **Practicality**: Validation through test cases confirms the application's ability to handle blockchain workflows in enterprise environments. The project underscores the potential of permissioned blockchains for secure, auditable asset management