ASSIGNMENT NO. 1

AIM : To understand the working mechanism of blockchain transactions and the concept of Distributed Ledger Technology (DLT) through an interactive simulation provided by Anders Brownworth.

Introduction:

Blockchain technology is revolutionizing the way digital transactions are recorded and verified. Unlike traditional centralized systems, blockchain operates on a decentralized network where data integrity and transparency are paramount. One of the core components of blockchain is the use of Distributed Ledger Technology (DLT), which ensures that all participants in the network share and synchronize the same data. This assignment explores the core functionalities of blockchain transactions and DLT using an interactive demo created by Anders Brownworth.

Theory:

1. Blockchain Transactions:

Blockchain transactions involve the transfer of digital assets from one party to another, recorded in a series of connected blocks. Each transaction is verified, encrypted, and stored in a block. These blocks are linked using cryptographic hashes, forming an immutable chain that grows with each new transaction.

2. Hashing and Block Integrity:

Each block has a unique hash generated based on its contents. Any modification to the block data changes its hash, invalidating the chain. This mechanism ensures data security and integrity. Each block also includes the hash of the previous block, thus forming a secure chain from the first (genesis) block to the latest.

3. Distributed Ledger Technology (DLT):

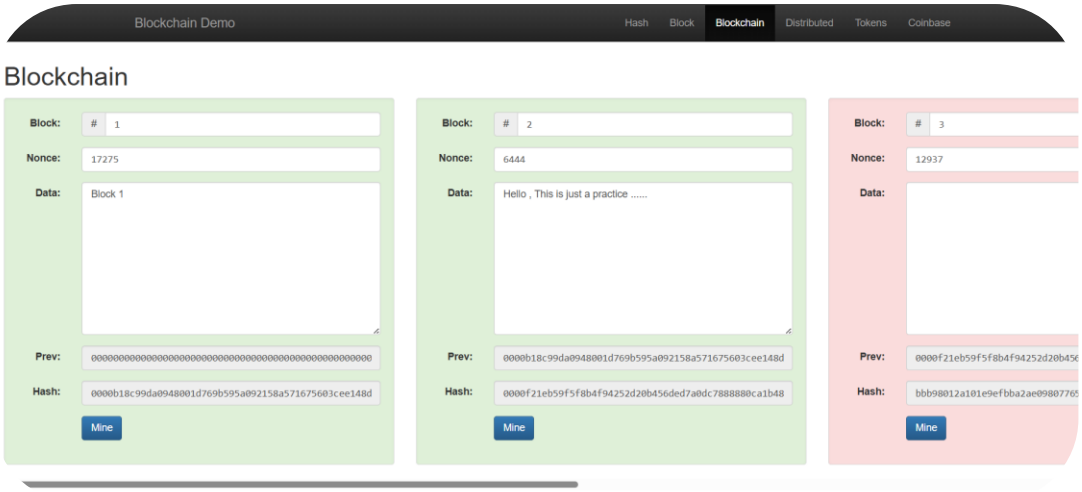
DLT refers to the system where the ledger (record of transactions) is distributed across all participants in the network. Every participant maintains a copy of the ledger, and changes are reflected across all copies after consensus. This prevents data tampering and promotes trust.

4. Proof of Work (PoW):

PoW is a consensus algorithm used to validate transactions and secure the network. It involves solving complex mathematical puzzles, which requires time and computational power. Once solved, the block is mined and added to the chain.

Activity Using the Demo (https://andersbrownworth.com/blockchain):

1. Open the website and go to the **Blockchain** section.
2. Observe the structure of blocks: Index, Nonce, Data, Hash, and Previous Hash.
3. Modify the data of a block and observe how the hash changes and subsequent blocks turn red to indicate invalidity.
4. Click on **"Mine"** to generate a valid hash (one that starts with four zeroes), restoring the chain's integrity.
5. Repeat the process for different blocks to understand how each one is connected and affected by changes.
6. Visit the **Distributed** section to see how different nodes hold copies of the ledger. Make changes in one and see how others reflect the updates upon achieving consensus.



Conclusion:

Blockchain and Distributed Ledger Technology offer a secure, transparent, and decentralized approach to managing digital transactions. Through this assignment and the hands-on demo, it is evident that blockchain's integrity relies on cryptographic hashing and consensus mechanisms. DLT ensures that no single party can manipulate the data, as the ledger is shared and verified by all network participants. This practical understanding lays a strong foundation for further exploration into blockchain-based applications in finance, governance, and beyond.

ASSIGNMENT NO. 2

AIM : To implement a program that converts a given input text into a hash using the SHA-256 algorithm.

Introduction:

In the modern digital world, data security is paramount. One of the most widely used cryptographic hash functions is **SHA-256 (Secure Hash Algorithm 256-bit)**, part of the SHA-2 family. It is used extensively in blockchain technology, digital signatures, and password security. This assignment focuses on implementing SHA-256 to convert any input text into a fixed-length, 256-bit hash.

What is SHA-256?

SHA-256 is a **one-way cryptographic function** that takes an input (or message) and returns a fixed 256-bit (32-byte) hash value. The resulting hash is unique to the input data—any change in the input will produce an entirely different hash. It is deterministic (same input always gives the same hash) and irreversible (hash cannot be converted back to original input).

Applications of SHA-256:

- Securing passwords
- Blockchain mining and transaction integrity
- Digital signatures and certificate verification
- Data integrity checks

Program Implementation :

```
import hashlib

# Input from user

text = input("Enter text to hash using SHA-256: ")

# Encoding the text to bytes

encoded_text = text.encode()

# Creating SHA-256 hash object

hash_object = hashlib.sha256(encoded_text)
```

```
# Getting the hexadecimal representation of the hash
hash_hex = hash_object.hexdigest()
print("SHA-256 Hash:", hash_hex)
```

Output

```
Enter text to hash using SHA-256: blockchain
SHA-256 Hash: ef7797e13d3a75526946a3bcf00d887f0da01d988c5c2bf37fdd5ec8c4d2aee0
```

Observations:

- The length of the SHA-256 output is always 64 characters in hexadecimal (256 bits).
- Even a single character change in the input text leads to a totally different hash.
- SHA-256 is a fast and secure method for ensuring data integrity.
- The hashing process is non-reversible, making it suitable for storing sensitive data like passwords securely.

Conclusion:

In this assignment, we successfully implemented a program that takes user input and converts it to a **SHA-256 hash** using Python. This practical application demonstrates the effectiveness of cryptographic hashing in securing and validating data. Understanding hashing functions like SHA-256 is crucial for fields like cybersecurity, blockchain development, and software engineering.

ASSIGNMENT NO. 3

AIM : To create and execute a simple Ethereum wallet transaction from one account to another using MetaMask.

Introduction:

With the rise of decentralized applications (dApps), interacting with the Ethereum blockchain has become easier using wallet extensions like **MetaMask**. MetaMask is a browser-based wallet that allows users to manage Ethereum accounts, send/receive cryptocurrency, and interact with smart contracts directly from their browser.

In this assignment, we explore how to create a simple transaction—transferring Ether from one account to another—using MetaMask.

What is MetaMask?

MetaMask is a **non-custodial cryptocurrency wallet** that provides access to the Ethereum blockchain via browser extensions and mobile apps. It enables users to store keys, manage accounts, and sign blockchain transactions securely.

Pre-requisites:

- MetaMask extension installed in your browser (Chrome, Brave, or Firefox)
- Two Ethereum accounts (one sender, one receiver)
- Some test ETH (e.g., from a faucet) if working on a testnet like Goerli

Procedure:

1. Set Up MetaMask:

- Install MetaMask from https://metamask.io/
- Create or import a wallet
- Choose a test network (e.g., **Goerli Testnet**)
- Copy your public wallet address (Account 1)

2. Fund the Wallet (Testnet):

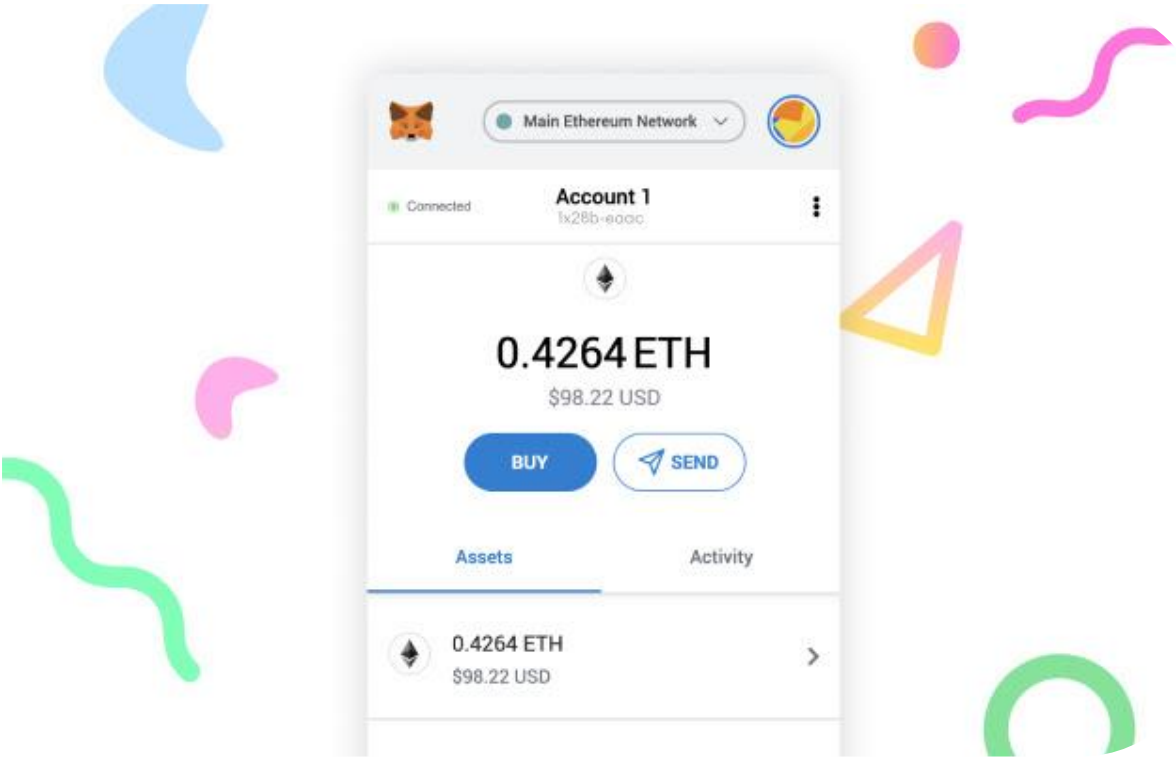
- Use a faucet (e.g., https://goerlifaucet.com/) to request test ETH
- Ensure Account 1 has a balance

3. Create the Transaction:

- Click **"Send"** in MetaMask
- Enter the recipient's address (Account 2)
- Enter the amount of ETH to send (e.g., 0.01 ETH)
- Choose transaction fee (use default or custom gas)
- Click **"Next"** and then **"Confirm"**

4. Verify the Transaction:

- After confirmation, go to https://goerli.etherscan.io/
- Paste the sender or receiver address to view transaction details
- Check status, gas used, and block confirmation



Conclusion:

This assignment successfully demonstrated how to perform a simple Ethereum transaction using **MetaMask**. The process emphasizes key aspects of blockchain transactions: address management, gas fees, and transaction verification. MetaMask serves as an essential tool for developers and users interacting with dApps and smart contracts.

ASSIGNMENT NO. 4

AIM : To connect MetaMask with a locally hosted Ganache test network and enable transaction testing in a private Ethereum blockchain environment.

Introduction:

Ganache is a personal Ethereum blockchain used for testing and development purposes. It provides a local blockchain with pre-funded accounts, allowing developers to simulate transactions, deploy smart contracts, and debug without spending real Ether. MetaMask, on the other hand, is a popular Ethereum wallet that can be configured to connect to custom networks like Ganache. This assignment walks through the process of linking MetaMask with Ganache.

What is Ganache?

Ganache, part of the Truffle Suite, creates a **local Ethereum blockchain**. It runs on your machine and provides 10 accounts with private keys and a default Ether balance, ideal for testing smart contracts and dApps.

What is MetaMask?

MetaMask is a **browser extension wallet** for Ethereum and compatible networks. It allows users to manage identities, send/receive Ether, and interact with deployed smart contracts via a simple interface.

Pre-requisites:

- MetaMask installed in a web browser
- Ganache installed on your local system (Download from: https://trufflesuite.com/ganache)
- Node.js and Truffle (optional for smart contract testing)

Procedure:

1. Launch Ganache:

- Open Ganache on your local machine
- Note the RPC Server URL (typically http://127.0.0.1:7545)
- Note one of the account addresses and its private key

2. Open MetaMask:

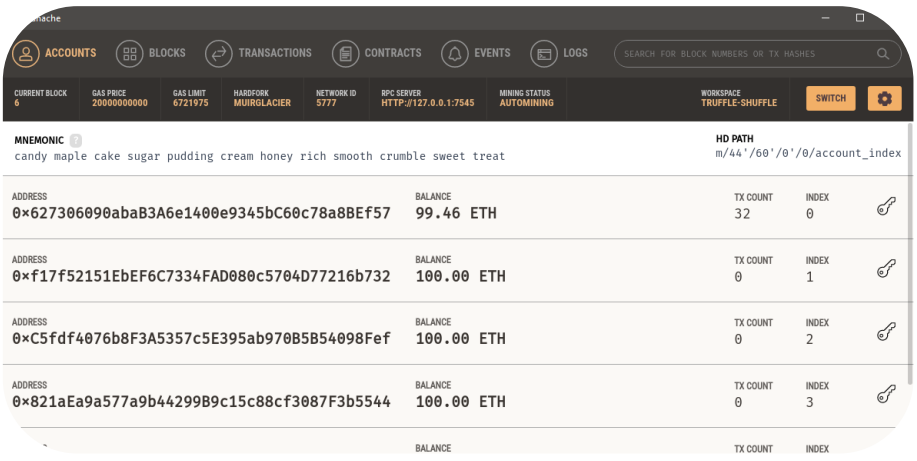
- Open your MetaMask extension
- Click on the **network dropdown** (usually showing "Ethereum Mainnet")

3. Add a Custom Network:

- Click **"Add network"**
- Fill in the following details:
    - **Network Name:** Ganache Local
    - **New RPC URL:** http://127.0.0.1:7545
    - **Chain ID:** 1337 (or 5777 depending on Ganache version)
    - **Currency Symbol (Optional):** ETH
- Click **"Save"**

4. Import a Ganache Account into MetaMask:

- In Ganache, copy the **private key** of any account
- In MetaMask, click on the **Account icon > Import Account**
- Paste the private key and click **"Import"**
- The account with test ETH should now appear in MetaMask



Conclusion:

In this assignment, we successfully connected MetaMask to a locally hosted Ethereum blockchain using Ganache. This integration allows developers to simulate and test Ethereum transactions in a risk-free environment. Understanding this connection is fundamental for dApp development, enabling rapid prototyping and debugging without the need to spend actual cryptocurrency.

ASSIGNMENT NO. 5

AIM : To perform an Ether transaction between two accounts using the Ganache local blockchain environment.

Introduction:

Ganache is a powerful tool for Ethereum developers that allows for local blockchain simulation. It provides pre-funded Ethereum accounts, enabling users to test smart contracts and transactions without spending real ETH. In this assignment, we demonstrate how to send Ether from one account to another using Ganache, simulating real blockchain activity in a safe and cost-free environment.

What is an Ether Transaction?

An Ether transaction refers to the transfer of ETH from one Ethereum wallet address to another. Every transaction includes:

- Sender Address
- Receiver Address
- Amount
- Gas Fee
- Nonce
- Signature

On Ganache, these components behave just like on the mainnet, but faster and without actual cost.

Pre-requisites:

- Ganache installed and running locally
- Access to two Ethereum accounts from Ganache
- MetaMask connected to Ganache or a terminal with web3 or ethers.js

Procedure (Using MetaMask):

1. Launch Ganache:

- Open Ganache on your computer
- You will see 10 accounts with 100 ETH each by default

2. Import Ganache Accounts into MetaMask:

- Copy the **private key** of Account 1 from Ganache
- Open MetaMask → Click profile icon → **Import Account**
- Paste the private key and import the account
- Repeat for Account 2 if needed

3. Send Ether Transaction:

- In MetaMask, select **Account 1**
- Click **"Send"**
- Paste the address of **Account 2**
- Enter amount (e.g., 1 ETH)
- Click **"Next"** → Then **"Confirm"**

4. Verify Transaction:

- Check the transaction status in MetaMask
- In Ganache, balances of both accounts will update
- You can also view transaction details under the **"Transactions" tab** in Ganache

Conclusion:

In this assignment, we successfully executed an **Ether transaction using Ganache**. The transaction mimicked a real-world blockchain environment but operated within a local, safe, and controlled setting. This is invaluable for developers learning Ethereum, building dApps, or testing smart contracts. It highlights the importance of test networks for efficient development and secure experimentation.

ASSIGNMENT NO. 6

AIM : To write, compile, and understand a basic "Hello World" smart contract using **Solidity**, the primary programming language for developing smart contracts on the Ethereum blockchain.

Introduction:

Solidity is a statically typed, contract-oriented programming language designed specifically for writing smart contracts on Ethereum. Like traditional programs, smart contracts in Solidity contain logic, state variables, and functions. In this assignment, we explore the structure and syntax of Solidity by creating a simple **"Hello World"** contract.

This forms the foundational step for anyone diving into Ethereum dApp development.

What is a Smart Contract?

A smart contract is a **self-executing code** stored on the blockchain. It runs automatically when predefined conditions are met and is immutable once deployed.

Solidity smart contracts typically include:

- State Variables
- Functions (public/private/view/pure)
- Events
- Modifiers (optional)

Pre-requisites:

- A code editor like Remix IDE
- Basic knowledge of Ethereum and Solidity syntax
- No gas or wallet required for testing in Remix

Procedure:

1. Open Remix IDE:

- Visit https://remix.ethereum.org
- In the file explorer, create a new file: HelloWorld.sol

2. Write the Smart Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract HelloWorld {
    string public greeting = "Hello, World!";
    // Function to return the greeting
    function getGreeting() public view returns (string memory) {
        return greeting;
    }
}
```

3. Compile the Code:

- Go to the **Solidity Compiler** tab in Remix
- Select version 0.8.0 or higher and click **Compile HelloWorld.sol**

4. Deploy the Contract:

- Go to the **Deploy & Run Transactions** tab
- Select **JavaScript VM** as environment (no wallet needed)
- Click **Deploy**

5. Interact with the Contract:

- After deployment, the contract will appear under "Deployed Contracts"
- Click getGreeting() → You will see the output: "Hello, World!"

Conclusion:

In this assignment, we successfully wrote and deployed a simple **"Hello World"** smart contract using **Solidity**. It introduced the basic structure of a Solidity contract, including state variables and public view functions. Understanding this basic framework is essential for building more complex smart contracts and decentralized applications in the Ethereum ecosystem.

ASSIGNMENT NO. 7

AIM : To develop a simple smart contract for User Identity Management using the Solidity programming language, enabling decentralized storage and retrieval of basic user information on the Ethereum blockchain.

Introduction:

In decentralized applications (dApps), user identity management is crucial for ensuring secure, permissioned access and data integrity. Traditional identity systems store user details on centralized servers, making them vulnerable to breaches. Using **Solidity**, we can create a **smart contract** that stores user identities in a decentralized, tamper-proof manner on the blockchain.

What is User Identity Management?

User identity management involves:

- Storing user details (name, age, email, etc.)
- Associating user information with a wallet address
- Allowing users to register and update their info securely

With blockchain, identity information is linked to a wallet (Ethereum address), allowing for transparent and immutable identity handling.

Pre-requisites:

- Basic understanding of Solidity
- Access to Remix IDE
- JavaScript VM or injected web3 environment (e.g., MetaMask)

Procedure:

1. Open Remix IDE:

- Go to https://remix.ethereum.org
- Create a new file: UserIdentity.sol

2. Write the Smart Contract Code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract UserIdentity {
    struct User {
        string name;
        uint age;
        string email;
        bool isRegistered;
    }
    mapping(address => User) public users;
    // Register or update a user's identity
    function register(string memory _name, uint _age, string memory _email) public {
        users[msg.sender] = User(_name, _age, _email, true);
    }
    // Retrieve user identity
    function getUser(address _user) public view returns (string memory, uint, string memory) {
        require(users[_user].isRegistered, "User not registered.");
        User memory u = users[_user];
        return (u.name, u.age, u.email);
    }
}
```

3. Compile and Deploy:

- Compile with Solidity 0.8.0+

Conclusion:

In this assignment, we developed a basic **User Identity Management smart contract** in Solidity. The contract allows users to register their identity details and retrieve them based on their Ethereum wallet address. This approach ensures decentralized, secure, and immutable identity storage—one of the key features of blockchain-based systems.

ASSIGNMENT NO. 8

AIM : To design and implement a Crowdfunding smart contract using ERC-20 tokens in Solidity that allows users to create campaigns, pledge tokens, claim funds if goals are met, or withdraw tokens if the campaign fails.

Introduction:

Crowdfunding is a process of raising funds from multiple users for a specific goal or cause. Implementing this on blockchain ensures transparency, automation, and trust. Using **ERC-20 tokens** instead of Ether adds flexibility and allows token-based economies to fund projects. This contract demonstrates how campaigns can be created, pledged to, and resolved based on the funding goal.

Smart Contract Features:

1. Campaign creation by a user.
2. Pledging of ERC-20 tokens by multiple users.
3. Claiming tokens by campaign owner if the funding goal is met.
4. Allowing refund/withdrawal if the campaign fails.

Pre-requisites:

- ERC-20 Token Contract (already deployed or mock token)
- Remix IDE
- MetaMask or JavaScript VM
- Basic knowledge of Solidity and ERC-20 standards

Procedure:

1. **Create File CrowdFund.sol in Remix IDE:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
interface IERC20 {
    function transferFrom(address from, address to, uint value) external returns (bool);
    function transfer(address to, uint value) external returns (bool);
}
```

```
contract CrowdFund {
    struct Campaign {
        address creator;
        uint goal;
        uint pledged;
        uint32 startAt;
        uint32 endAt;
        bool claimed;
    }
    IERC20 public immutable token;
    uint public count;
    mapping(uint => Campaign) public campaigns;
    mapping(uint => mapping(address => uint)) public pledgedAmount;
    constructor(address _token) {
        token = IERC20(_token);
    }
    function createCampaign(uint _goal, uint32 _duration) external {
        require(_duration <= 90 days, "Too long");
        count += 1;
        campaigns[count] = Campaign({
            creator: msg.sender,
            goal: _goal,
            pledged: 0,
            startAt: uint32(block.timestamp),
            endAt: uint32(block.timestamp + _duration),
            claimed: false
        });
    }

    function pledge(uint _id, uint _amount) external {
```

```
    Campaign storage campaign = campaigns[_id];
        require(block.timestamp >= campaign.startAt, "Not started");
        require(block.timestamp <= campaign.endAt, "Ended");
        campaign.pledged += _amount;
        pledgedAmount[_id][msg.sender] += _amount;
        token.transferFrom(msg.sender, address(this), _amount);
    }
    function claim(uint _id) external {
        Campaign storage campaign = campaigns[_id];
        require(msg.sender == campaign.creator, "Not creator");
        require(block.timestamp > campaign.endAt, "Not ended");
        require(campaign.pledged >= campaign.goal, "Goal not met");
        require(!campaign.claimed, "Already claimed");
        campaign.claimed = true;
        token.transfer(campaign.creator, campaign.pledged);
    }
    function refund(uint _id) external {
        Campaign storage campaign = campaigns[_id];
        require(block.timestamp > campaign.endAt, "Not ended");
        require(campaign.pledged < campaign.goal, "Goal met");
        uint bal = pledgedAmount[_id][msg.sender];
        pledgedAmount[_id][msg.sender] = 0;
        token.transfer(msg.sender, bal);
    }
}
```

Observations:

Action	Result
Create Campaign	New campaign with goal and deadline
Pledge Tokens	Tokens transferred from user to contract
Claim Funds	Only if goal is met and campaign ends
Refund Tokens	Allowed if goal is not met after deadline
Security Checks	Valid creator, no double claim/refund

Conclusion:

In this assignment, we developed a fully functional crowdfunding smart contract using ERC-20 tokens. The contract ensures fair and trustless handling of campaign funds through goal-based fund release or refunds. This model demonstrates a decentralized alternative to platforms like Kickstarter, combining transparency with automation.

ASSIGNMENT NO. 9

AIM : To develop a blockchain-based NFT (Non-Fungible Token) application that enables fan engagement and distributes gaming rewards through digital collectibles using smart contracts on Ethereum.

Introduction:

NFTs (Non-Fungible Tokens) have revolutionized how digital assets are owned and traded, especially in fan communities and gaming ecosystems. NFTs represent unique digital items—such as collectibles, in-game assets, or membership perks—stored on the blockchain. This project focuses on building an NFT-based application to boost fan engagement and reward players or users through tokenized assets.

Use Case Overview:

The NFT app allows:

- Fans to **mint and collect unique NFTs**
- Gamers to **earn NFT rewards** based on achievements
- Community engagement through **token-based incentives**

This enhances interaction, promotes loyalty, and offers transparent reward distribution in games or fan platforms.

Technology Stack:

- **Solidity** – Smart contract development
- **ERC-721** – NFT standard
- **Remix IDE** or **Hardhat** – Development and deployment
- **MetaMask** – Wallet integration
- **JavaScript/HTML** – Basic frontend (optional)

NFT Smart Contract

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

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
```

```
import "@openzeppelin/contracts/access/Ownable.sol";

contract FanGameNFT is ERC721URIStorage, Ownable {

    uint public tokenCount;

    constructor() ERC721("FanGameNFT", "FGN") {}

    function mintNFT(address recipient, string memory tokenURI) public onlyOwner returns (uint) {

        tokenCount += 1;

        _mint(recipient, tokenCount);

        _setTokenURI(tokenCount, tokenURI);

        return tokenCount;

    }

}
```

Explanation:

- Uses OpenZeppelin's secure implementation of **ERC-721**.
- mintNFT allows the app/game admin (owner) to mint unique tokens for fans or gamers.
- tokenURI stores metadata (images, videos, badges, etc.).

Steps to Build the App:

1. **Set up development environment:**
    - Use Remix or Hardhat for smart contract writing and testing.
    - Install OpenZeppelin contracts via npm if using Hardhat.
2. **Write and deploy smart contract:**
    - Use the FanGameNFT contract shown above.
    - Deploy to testnet (e.g., Goerli) or local blockchain (Ganache/Hardhat).
3. **Mint NFTs for fans or gamers:**
    - Call mintNFT() after a user completes a milestone or joins an event.
    - Attach metadata using IPFS or any NFT metadata hosting.

4. **Frontend Integration (Optional):**
    - Create a simple HTML/JS interface to display NFTs, show wallet connection, and list achievements.

Use Cases & Benefits:

Feature	Benefit
NFT-based fan badges	Boosts community loyalty
In-game NFT items	Tradeable and ownable gaming rewards
Tokenized achievements	Immutable proof of skill or support
Interoperability with marketplaces	NFTs can be sold or transferred on OpenSea

Conclusion:

The NFT application empowers fan communities and gamers through **blockchain-based digital ownership**. It bridges the gap between creators and their supporters by offering collectible, tradable, and verifiable digital rewards. This opens up a new realm of possibilities in fan engagement and gaming ecosystems, encouraging more interactive, reward-driven experiences.

ASSIGNMENT NO. 10

AIM : To implement blockchain-based smart contracts and decentralized applications (DApps) for solving real-world problems in two critical domains:

- Tracking property details in real estate
- Protecting sensitive medical data in healthcare

These solutions ensure transparency, security, and trust using decentralized technology.

1. Blockchain in Real Estate: Property Tracking System

Introduction:

The real estate sector suffers from issues like forgery, illegal transfers, and lack of proper documentation. These challenges can be resolved using **blockchain** which provides immutable, time-stamped, and transparent records of property transactions.

By leveraging smart contracts, real estate properties can be **digitally registered, verified, and transferred** securely, eliminating intermediaries and reducing paperwork.

Features of the Smart Contract:

- Register new properties with location and price
- Store the current owner's wallet address
- Allow only the owner to transfer ownership
- Enable public viewing of property details for transparency

Smart Contract Code (Solidity):

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

pragma solidity ^0.8.0;

contract RealEstateRegistry {

    struct Property {

        uint id;

        string location;

        uint price;

        address owner;
```

```
    }
    uint public propertyCount;
    mapping(uint => Property) public properties;
    event PropertyRegistered(uint id, address indexed owner);
    event OwnershipTransferred(uint id, address indexed newOwner);
    function registerProperty(string memory _location, uint _price) public {
        propertyCount++;
        properties[propertyCount] = Property(propertyCount, _location, _price, msg.sender);
        emit PropertyRegistered(propertyCount, msg.sender);
    }
    function transferOwnership(uint _id, address _newOwner) public {
        require(msg.sender == properties[_id].owner, "Unauthorized");
        properties[_id].owner = _newOwner;
        emit OwnershipTransferred(_id, _newOwner);
    }
    function getProperty(uint _id) public view returns (string memory, uint, address) {
        Property memory p = properties[_id];
        return (p.location, p.price, p.owner);
    }
}
```

Benefits:

- Eliminates fake property claims
- Verifiable and permanent records
- Reduces cost and time of property transfer
- Enhances public trust in property dealings

2. Blockchain in Healthcare: DApp for Medical Data Protection

Introduction:

Medical data is extremely sensitive and often stored in centralized systems, vulnerable to breaches and unauthorized access. Blockchain provides a decentralized, access-controlled mechanism to store and retrieve encrypted medical records, giving patients full ownership and control.

Proposed Solution:

- Use IPFS (InterPlanetary File System) to store encrypted medical files.
- Store IPFS hashes and ownership metadata in a smart contract.
- Patients can control who accesses their medical data via wallet verification.
- Doctors or hospitals can upload data with patient consent.

Smart Contract Code (Solidity):

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

pragma solidity ^0.8.0;

contract MedicalDataVault {

    struct Record {

        string ipfsHash;

        address uploader;

        address patient;

    }

    uint public recordCount;

    mapping(uint => Record) public records;

    event RecordUploaded(uint id, address indexed uploader, address indexed patient);

    function uploadRecord(string memory _ipfsHash, address _patient) public {

        recordCount++;

        records[recordCount] = Record(_ipfsHash, msg.sender, _patient);

        emit RecordUploaded(recordCount, msg.sender, _patient);

    }

    function getRecord(uint _id) public view returns (string memory) {

        require(msg.sender == records[_id].patient, "Access denied");

        return records[_id].ipfsHash  }}
```

DApp Workflow:

1. **Doctor uploads encrypted data** → Sends IPFS hash to contract.
2. **Contract stores** uploader and patient address with metadata.
3. **Patient logs in via MetaMask** and accesses only their data.
4. **No centralized storage** → Enhanced privacy and security.

Security & Privacy Highlights:

- Blockchain ensures **immutability** and **traceability** of data.
- IPFS provides decentralized file hosting.
- Data access is **wallet-based**, eliminating misuse.
- Only authorized patients can read/download records.

Combined Impact of Blockchain in Both Domains

Feature	Real Estate	Healthcare
Ownership Tracking	Immutable property records	Verifiable patient identity
Access Control	Only owner can transfer	Only patient can access medical data
Fraud Prevention	Eliminates fake registrations	Prevents unauthorized data sharing
Transparency	Publicly verifiable records	Logs every access and upload
Cost and Time Efficiency	No middlemen, less paperwork	Instant data sharing and verification

Conclusion:

Blockchain offers an innovative and powerful approach to managing sensitive, high-value information. Through smart contracts and decentralized apps:

- Real estate becomes more **trustworthy and fraud-resistant**.
- Healthcare evolves into a **patient-controlled, privacy-centric** system.

These blockchain solutions pave the way for a **transparent, secure, and decentralized .**

ASSIGNMENT NO. 11

AIM : To design and implement an NFT-based English Auction system using Solidity. The application will enable a seller to auction their NFT for a period of 7 days, allowing participants to place bids using ETH. The auction ensures fairness by allowing only higher bids and supporting bid withdrawal for non-winning bidders.

1. Introduction

What is an English Auction?

An English auction is a type of auction where participants place increasingly higher bids. The auction ends after a fixed duration, and the highest bidder wins the item. In this case, the item is an NFT (Non-Fungible Token).

Why Use Blockchain for Auctions?

Traditional auctions often rely on centralized intermediaries, leading to transparency and trust issues. By leveraging Ethereum smart contracts, we can:

- Ensure fair bidding without manipulation.
- Allow real-time tracking of bids.
- Automate fund transfer and ownership transfer.

2. Smart Contract Features

Roles:

- **Seller:** Deploys the auction contract and owns the NFT.
- **Bidders:** Can participate by sending ETH bids.
- **Highest Bidder:** Becomes the new NFT owner after auction ends.

Functional Requirements:

- Auction duration is 7 days from deployment.
- Only bids higher than the current highest bid are accepted.
- Outbid users can withdraw their ETH.
- On completion, NFT is transferred to the highest bidder and ETH is sent to the seller.

2. Solidity Smart Contract (Core)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
interface INFT {
    function transferFrom(address from, address to, uint tokenId) external;
}
contract EnglishAuction {
    address payable public seller;
    uint public endAt;
    bool public started;
    bool public ended;
    INFT public nft;
    uint public nftId;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) public bids;
    constructor(address _nft, uint _nftId) {
        seller = payable(msg.sender);
        nft = INFT(_nft);
        nftId = _nftId;
    }
    function start() external {
        require(msg.sender == seller, "Only seller can start");
        require(!started, "Already started");
        started = true;
        endAt = block.timestamp + 7 days;
        nft.transferFrom(msg.sender, address(this), nftId);
    }
    function bid() external payable {
        require(started, "Auction not started");
```

```
        require(block.timestamp < endAt, "Auction ended");
        require(msg.value > highestBid, "Bid too low");
        if (highestBidder != address(0)) {
            bids[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    function withdraw() external {
        uint amount = bids[msg.sender];
        require(amount > 0, "No funds to withdraw");
        bids[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
    function end() external {
        require(started, "Auction not started");
        require(block.timestamp >= endAt, "Auction not yet ended");
        require(!ended, "Auction already ended");
        ended = true;
        if (highestBidder != address(0)) {
            nft.transferFrom(address(this), highestBidder, nftId);
            seller.transfer(highestBid);
        } else {
            nft.transferFrom(address(this), seller, nftId);
        }
    }
}
```

4. Workflow of the Auction DApp

1. **NFTDeployment:**
   The seller creates an NFT and approves the auction contract to transfer it.
2. **StartAuction:**
   Seller calls start() to transfer the NFT to the contract and begin the 7-day auction.
3. **PlaceBids:**
   Participants place bids by sending ETH using bid(). Each bid must exceed the current highest bid.
4. **WithdrawOutbidETH:**
   Users who have been outbid can call withdraw() to retrieve their ETH safely.
5. **EndAuction:**
   After 7 days, end() finalizes the auction. The highest bidder receives the NFT, and the seller receives the ETH.

5. Benefits & Security

Benefits:

- Transparent and automatic bid processing
- Trustless transfer of assets and ETH
- No need for centralized auction platforms

Security Considerations:

- Prevent reentrancy in withdraw function (can use ReentrancyGuard).
- Ensure NFT approval before transferring.
- Handle refunds safely to prevent locked ETH.

Conclusion:

This project demonstrates how blockchain can automate and secure the auction process using NFTs and smart contracts. The English Auction model provides a fair, transparent mechanism for exchanging digital assets. By implementing it with Solidity and integrating with frontends like MetaMask and Web3.js, we can create a fully decentralized auction experience.

ASSIGNMENT NO. 12

AIM : To design and implement a permissioned business network using Hyperledger Fabric, demonstrating how organizations can securely and transparently share data and transactions within a decentralized ledger system.

1. Introduction

What is Hyperledger Fabric?

Hyperledger Fabric is a modular and extensible permissioned blockchain platform designed for enterprise use. Unlike public blockchains (like Ethereum or Bitcoin), Fabric allows only authorized participants to access the network, making it ideal for businesses that require privacy, confidentiality, and fine-grained control over data sharing.

Key Concepts:

- **Permissioned Network:** Only known and approved entities participate.
- **Chaincode (Smart Contract):** Contains the business logic.
- **Channels:** Private subnets between specific organizations.
- **Peers and Orderers:** Nodes that endorse and order transactions.
- **Certificate Authority (CA):** Issues digital identities to network participants.

2. Objectives

- Set up a basic Hyperledger Fabric network.
- Define a business network with two organizations.
- Deploy chaincode to simulate asset creation and transfer.
- Interact with the network using CLI or SDK.

3. Network Setup Architecture

Network Components:

- 2 Organizations (Org1 & Org2)
- 1 Ordering Service
- 1 Channel (mychannel)
- 1 Chaincode (e.g., assetTransfer)

Each organization has:

- 1 Peer node
- A CA for identity management
- MSP (Membership Service Provider)

4. Steps to Create the Business Network

Step 1: Prerequisites

- Install Docker & Docker Compose
- Install Node.js and npm
- Install Hyperledger Fabric samples and binaries

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Step 2: Generate Crypto Material

Use cryptogen or Fabric CA to generate identities (certificates and keys) for all network participants.

Step 3: Define the Network Configuration

Edit YAML files (docker-compose.yaml, configtx.yaml, core.yaml) to define:

- Organizations
- Channels
- Policies
- Genesis block

Step 4: Start the Network

Launch the network using Docker:

```
docker-compose -f docker-compose.yaml up -d
```

Step 5: Create Channel

```
peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel.tx
```

Step 6: Deploy Chaincode

Install and approve the chaincode for both organizations:

```
peer lifecycle chaincode install assettransfer.tar.gz
```

Then commit and invoke the chaincode to perform transactions.

5. Example Use Case: Asset Transfer

Chaincode Logic:

- CreateAsset: Adds a new asset to the ledger.
- ReadAsset: Retrieves asset details.
- TransferAsset: Changes the ownership of an asset.

Each transaction is recorded immutably, visible only to authorized participants.

6. Benefits of Using Hyperledger Fabric

- **Permissioned Access:** Only verified participants access the ledger.
- **Data Confidentiality:** Channel and private data collections protect sensitive data.
- **Modular Design:** Pluggable consensus and components.
- **High Performance:** Supports parallel execution and scalability.

Conclusion

This assignment demonstrates how Hyperledger Fabric can be used to build a secure and transparent business network. Organizations can collaborate while maintaining control over their data. By understanding the architecture and tools, developers can build enterprise-grade blockchain applications suited for industries like supply chain, healthcare, and finance.