**E-voting**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract EVoting {

address public owner;

mapping(address => bool) public hasVoted;

mapping(bytes32 => uint256) public votes;

enum VotingState { NotStarted, InProgress, Ended }

VotingState public votingState;

event VoteCasted(address indexed voter, bytes32 indexed candidate);

constructor() {

owner = msg.sender;

votingState = VotingState.NotStarted;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can perform this operation");

\_;

}

modifier onlyDuringVoting() {

require(votingState == VotingState.InProgress, "Voting is not in progress");

\_;

}

modifier hasNotVoted() {

require(!hasVoted[msg.sender], "You have already voted");

\_;

}

modifier isValidCandidate(bytes32 candidate) {

require(candidate != bytes32(0), "Invalid candidate");

\_;

}

function startVoting() public onlyOwner {

require(votingState == VotingState.NotStarted, "Voting has already started");

votingState = VotingState.InProgress;

}

function endVoting() public onlyOwner {

require(votingState == VotingState.InProgress, "Voting is not in progress");

votingState = VotingState.Ended;

}

function vote(bytes32 candidate)

public

onlyDuringVoting

hasNotVoted

isValidCandidate(candidate)

{

hasVoted[msg.sender] = true;

votes[candidate] += 1;

emit VoteCasted(msg.sender, candidate);

}

function getVoteCount(bytes32 candidate) public view isValidCandidate(candidate) returns (uint256) {

return votes[candidate];

}

function checkVotingState() public view returns (VotingState) {

return votingState;

}

}

**educational certificates**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract EducationalCertificateRegistry {

address public owner;

struct Certificate {

address owner;

string institution;

string degree;

uint256 graduationYear;

}

mapping(address => Certificate) public certificates;

event CertificateStored(address indexed owner, string institution, string degree, uint256 graduationYear);

constructor() {

owner = msg.sender;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can perform this operation");

\_;

}

function storeCertificate(string memory institution, string memory degree, uint256 graduationYear) public {

require(bytes(institution).length > 0, "Institution name cannot be empty");

require(bytes(degree).length > 0, "Degree name cannot be empty");

require(graduationYear > 1900, "Invalid graduation year");

certificates[msg.sender] = Certificate(msg.sender, institution, degree, graduationYear);

emit CertificateStored(msg.sender, institution, degree, graduationYear);

}

function getCertificate(address certificateOwner) public view returns (string memory institution, string memory degree, uint256 graduationYear) {

Certificate storage certificate = certificates[certificateOwner];

return (certificate.institution, certificate.degree, certificate.graduationYear);

}

}

**Health Record**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract HealthRecordRegistry {

address public owner;

struct HealthRecord {

address patient;

string data;

}

mapping(address => HealthRecord) public healthRecords;

mapping(address => bool) public isPhysician;

mapping(address => bool) public isLabTechnician;

event HealthRecordStored(address indexed patient, string data);

event PhysicianAdded(address indexed physician);

event LabTechnicianAdded(address indexed labTechnician);

constructor() {

owner = msg.sender;

isPhysician[msg.sender] = true;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can perform this operation");

\_;

}

modifier onlyPatient() {

require(msg.sender == healthRecords[msg.sender].patient, "Only the patient can perform this operation");

\_;

}

modifier onlyPhysician() {

require(isPhysician[msg.sender], "Only physicians can perform this operation");

\_;

}

modifier onlyLabTechnician() {

require(isLabTechnician[msg.sender], "Only lab technicians can perform this operation");

\_;

}

function storeHealthRecord(string memory data) public {

require(bytes(data).length > 0, "Health record data cannot be empty");

healthRecords[msg.sender] = HealthRecord(msg.sender, data);

emit HealthRecordStored(msg.sender, data);

}

function getHealthRecord(address patient) public view returns (string memory data) {

require(msg.sender == patient || isPhysician[msg.sender] || isLabTechnician[msg.sender], "Permission denied");

HealthRecord storage record = healthRecords[patient];

return record.data;

}

function addPhysician(address physician) public onlyOwner {

isPhysician[physician] = true;

emit PhysicianAdded(physician);

}

function addLabTechnician(address labTechnician) public onlyOwner {

isLabTechnician[labTechnician] = true;

emit LabTechnicianAdded(labTechnician);

}

}

**supply chain management**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract SupplyChainManagement {

address public owner;

enum ShipmentStatus { Pending, InTransit, Delivered, Received }

struct Shipment {

address provider;

address manager;

address tracing;

ShipmentStatus status;

string product;

}

mapping(uint256 => Shipment) public shipments;

uint256 public shipmentCounter;

event ShipmentCreated(uint256 indexed shipmentId, address indexed provider, string product);

event ShipmentStatusUpdated(uint256 indexed shipmentId, ShipmentStatus status);

event ProductReceived(uint256 indexed shipmentId);

constructor() {

owner = msg.sender;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can perform this operation");

\_;

}

function createShipment(address manager, address tracing, string memory product) public {

shipmentCounter++;

shipments[shipmentCounter] = Shipment(msg.sender, manager, tracing, ShipmentStatus.Pending, product);

emit ShipmentCreated(shipmentCounter, msg.sender, product);

}

function updateShipmentStatus(uint256 shipmentId, ShipmentStatus newStatus) public {

Shipment storage shipment = shipments[shipmentId];

require(shipment.manager == msg.sender, "Only the shipment manager can update the status");

require(shipment.status < newStatus, "Invalid status update");

shipment.status = newStatus;

emit ShipmentStatusUpdated(shipmentId, newStatus);

}

function receiveProduct(uint256 shipmentId) public {

Shipment storage shipment = shipments[shipmentId];

require(shipment.tracing == msg.sender, "Only the tracing participant can confirm product receipt");

require(shipment.status == ShipmentStatus.Delivered, "Product can be received only after it's delivered");

shipment.status = ShipmentStatus.Received;

emit ProductReceived(shipmentId);

}

function getShipmentStatus(uint256 shipmentId) public view returns (ShipmentStatus) {

return shipments[shipmentId].status;

}

function getShipmentProduct(uint256 shipmentId) public view returns (string memory) {

return shipments[shipmentId].product;

}

}

**financial transactions**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract FinancialTransactions {

address public owner;

mapping(address => uint256) public balances;

constructor() {

owner = msg.sender;

}

function deposit() public payable {

balances[msg.sender] += msg.value;

}

function withdraw(uint256 amount) public {

require(balances[msg.sender] >= amount, "Insufficient balance");

balances[msg.sender] -= amount;

payable(msg.sender).transfer(amount);

}

}

**land registration system**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract LandRegistrationSystem {

address public owner;

enum LandStatus { PendingVerification, Verified, Transferred }

struct LandRecord {

address owner;

string location;

string documentHash; // Hash of land document for verification

LandStatus status;

}

mapping(uint256 => LandRecord) public landRecords;

uint256 public landCounter;

event LandRegistered(uint256 indexed landId, address indexed owner, string location);

event LandVerified(uint256 indexed landId);

event LandTransferred(uint256 indexed landId, address indexed newOwner);

constructor() {

owner = msg.sender;

}

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can perform this operation");

\_;

}

function registerLand(string memory location, string memory documentHash) public {

landCounter++;

landRecords[landCounter] = LandRecord(msg.sender, location, documentHash, LandStatus.PendingVerification);

emit LandRegistered(landCounter, msg.sender, location);

}

function verifyLandDocument(uint256 landId, string memory providedDocumentHash) public onlyOwner {

LandRecord storage land = landRecords[landId];

require(land.status == LandStatus.PendingVerification, "Land is not pending verification");

require(keccak256(bytes(providedDocumentHash)) == keccak256(bytes(land.documentHash)), "Document verification failed");

land.status = LandStatus.Verified;

emit LandVerified(landId);

}

function transferLandOwnership(uint256 landId, address newOwner) public {

LandRecord storage land = landRecords[landId];

require(land.status == LandStatus.Verified, "Land is not verified");

require(msg.sender == land.owner, "Only the current owner can transfer the land");

land.owner = newOwner;

land.status = LandStatus.Transferred;

emit LandTransferred(landId, newOwner);

}

}

**lottery system**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Lottery {

address public manager;

address[] public players;

address public winner;

modifier onlyManager() {

require(msg.sender == manager, "Only the manager can perform this operation");

\_;

}

constructor() {

manager = msg.sender;

}

function enter() public payable {

require(msg.value > 0.01 ether, "Minimum contribution is 0.01 ether");

players.push(msg.sender);

}

function getPlayers() public view returns (address[] memory) {

return players;

}

function pickWinner() public onlyManager {

require(players.length > 0, "No players in the lottery");

// Pseudo-randomly select a winner based on block information

uint index = random() % players.length;

winner = players[index];

// Transfer the entire balance to the winner

payable(winner).transfer(address(this).balance);

// Reset the players array for the next round

players = new address[](0);

}

function random() private view returns (uint) {

return uint(keccak256(abi.encodePacked(block.difficulty, block.timestamp, players)));

}

}