CAKE PREPARETION:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CakePreparation {

    struct Ingredient {

        string name;

        uint256 quantity;

    }

    mapping(uint256 => Ingredient) public ingredients;

    uint256 public ingredientCount;

    constructor() {

        ingredientCount = 0;

    }

    function addIngredient(string memory \_name, uint256 \_quantity) public {

        Ingredient storage newIngredient = ingredients[ingredientCount];

        newIngredient.name = \_name;

        newIngredient.quantity = \_quantity;

        ingredientCount++;

    }

    function getIngredient(uint256 \_index) public view returns (string memory, uint256) {

        require(\_index < ingredientCount, "Invalid ingredient index");

        return (ingredients[\_index].name, ingredients[\_index].quantity);

    }

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

        require(ingredientCount > 0, "No ingredients added");

        string memory preparationSteps = "Cake preparation steps:\n";

        for (uint256 i = 0; i < ingredientCount; i++) {

            (string memory name, uint256 quantity) = getIngredient(i);

            preparationSteps = string(abi.encodePacked(preparationSteps, "- Add ", toString(quantity), " ", name, "\n"));

        }

        return preparationSteps;

    }

    // Helper function to convert uint256 to string

    function toString(uint256 value) internal pure returns (string memory) {

        if (value == 0) {

            return "0";

        }

        uint256 tempValue = value;

        uint256 digits;

        while (tempValue > 0) {

            tempValue /= 10;

            digits++;

        }

        bytes memory buffer = new bytes(digits);

        for (uint256 i = 0; i < digits; i++) {

            buffer[digits - i - 1] = bytes1(uint8(48 + uint256(value / (10\*\*i)) % 10));

        }

        return string(buffer);

    }

}

CROWDFUNDING

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Crowdfunding {

    address public owner;

    uint public goal;

    uint public deadline;

    mapping(address => uint) public donations;

    uint public totalDonations;

    constructor(uint \_goal, uint \_durationDays) {

        owner = msg.sender;

        goal = \_goal;

        deadline = block.timestamp + (\_durationDays \* 1 days);

    }

    function donate() public payable {

        require(block.timestamp < deadline, "Crowdfunding has ended.");

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

        totalDonations += msg.value;

    }

    function withdrawFunds() public {

        require(msg.sender == owner, "Only the owner can withdraw the funds.");

        require(block.timestamp >= deadline, "Crowdfunding is still ongoing.");

        require(totalDonations >= goal, "Goal not reached.");

        payable(owner).transfer(totalDonations);

    }

    function getRefund() public {

        require(block.timestamp >= deadline, "Crowdfunding is still ongoing.");

        require(totalDonations < goal, "Goal has been reached.");

        uint donationAmount = donations[msg.sender];

        require(donationAmount > 0, "You have not donated.");

        donations[msg.sender] = 0;

        totalDonations -= donationAmount;

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

    }

}

ETHTRANSFER.SOL

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract EtherTransfer {

    function transferEther(address payable \_recipient) external payable {

        require(\_recipient != address(0), "Invalid recipient address");

        require(msg.value > 0, "Amount must be greater than 0");

        // Transfer Ether to the recipient

        \_recipient.transfer(msg.value);

    }

}

EVOTING

// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

contract EVoting {

    struct Candidate {

        uint256 id;

        string name;

        uint256 voteCount;

    }

    address public owner;

    uint256 public endTime;

    uint256 public candidatesCount;

    mapping(uint256 => Candidate) public candidates;

    mapping(address => bool) public hasVoted;

    constructor(uint256 \_timeperiod, uint256 \_candidatesCount, string[] memory \_names) {

        owner = msg.sender;

        endTime = block.timestamp + \_timeperiod;

        candidatesCount = \_candidatesCount;

        for (uint256 i = 0; i < candidatesCount; i++) {

            candidates[i] = Candidate(i, \_names[i], 0);

        }

    }

    function vote(uint256 \_candidateId) public {

        require(msg.sender != owner, "The owner's address cannot cast a vote!");

        require(!hasVoted[msg.sender], "This address has already voted!");

        require(\_candidateId < candidatesCount, "Invalid candidate ID!");

        require(block.timestamp < endTime, "Voting time is over!");

        hasVoted[msg.sender] = true;

        candidates[\_candidateId].voteCount++;

    }

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

        require(block.timestamp > endTime, "The E-Voting has not yet ended.");

        uint256 winningVoteCount = 0;

        uint256 winningCandidateId = 0;

        for (uint256 i = 0; i < candidatesCount; i++) {

            if (candidates[i].voteCount > winningVoteCount) {

                winningVoteCount = candidates[i].voteCount;

                winningCandidateId = i;

            }

        }

        return candidates[winningCandidateId].name;

    }

}

HEALTHCARE.SOL

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract Healthcare {

    address public owner;

    struct Patient {

        uint patientId;

        string name;

        uint age;

        uint[] appointments;

    }

    struct Doctor {

        uint doctorId;

        string name;

        string specialization;

        uint[] appointments;

    }

    struct Appointment {

        uint appointmentId;

        uint patientId;

        uint doctorId;

        uint date;

        bool isCompleted;

        string prescription;

    }

    mapping (uint => Patient) public patients;

    mapping (uint => Doctor) public doctors;

    mapping (uint => Appointment) public appointments;

    event AppointmentCreated(uint indexed appointmentId, uint indexed patientId, uint indexed doctorId);

    event AppointmentCompleted(uint indexed appointmentId, string prescription);

    modifier onlyOwner() {

        require(msg.sender == owner, "Only contract owner can perform this action");

        \_;

    }

    constructor() {

        owner = msg.sender;

    }

    function createPatient(string memory \_name, uint \_age) public returns (uint) {

        uint patientId = uint(keccak256(abi.encodePacked(block.timestamp, msg.sender)));

        patients[patientId] = Patient(patientId, \_name, \_age, new uint[](0));

        return patientId;

    }

    function createDoctor(string memory \_name, string memory \_specialization) public onlyOwner returns (uint) {

        uint doctorId = uint(keccak256(abi.encodePacked(block.timestamp, msg.sender)));

        doctors[doctorId] = Doctor(doctorId, \_name, \_specialization, new uint[](0));

        return doctorId;

    }

    function createAppointment(uint \_patientId, uint \_doctorId, uint \_date) public {

        Patient storage myPatient = patients[\_patientId];

        require(myPatient.patientId != 0, "Patient does not exist");

        Doctor storage myDoctor = doctors[\_doctorId];

        require(myDoctor.doctorId != 0, "Doctor does not exist");

        uint appointmentId = uint(keccak256(abi.encodePacked(block.timestamp, msg.sender, \_patientId, \_doctorId)));

        appointments[appointmentId] = Appointment(appointmentId, \_patientId, \_doctorId, \_date, false, "");

        myPatient.appointments.push(appointmentId);

        myDoctor.appointments.push(appointmentId);

        emit AppointmentCreated(appointmentId, \_patientId, \_doctorId);

    }

    function completeAppointment(uint \_appointmentId, string memory \_prescription) public {

        Appointment storage myAppointment = appointments[\_appointmentId];

        require(myAppointment.appointmentId != 0, "Appointment does not exist");

        require(msg.sender == doctors[myAppointment.doctorId].doctorId, "Only the doctor can complete the appointment");

        myAppointment.isCompleted = true;

        myAppointment.prescription = \_prescription;

        emit AppointmentCompleted(\_appointmentId, \_prescription);

    }

    function getPatient(uint \_patientId) public view returns (string memory, uint, uint[] memory) {

        Patient storage myPatient = patients[\_patientId];

        return (myPatient.name, myPatient.age, myPatient.appointments);

    }

    function getDoctor(uint \_doctorId) public view returns (string memory, string memory, uint[] memory) {

        Doctor storage myDoctor = doctors[\_doctorId];

        return (myDoctor.name, myDoctor.specialization, myDoctor.appointments);

    }

    function getAppointment(uint \_appointmentId) public view returns (uint, uint, uint, uint, bool, string memory

INSURANCE

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract Insurance {

    address public owner;

    uint public policyCounter;

    struct Policy {

        uint policyId;

        address policyHolder;

        uint premium;

        uint coverage;

        bool isActive;

        uint expirationDate;

        uint[] claimIds;

    }

    struct Claim {

        uint claimId;

        uint policyId;

        uint amount;

        string reason;

        bool isApproved;

        address[] approvers;

    }

    mapping(uint => Policy) public policies;

    mapping(uint => Claim) public claims;

    event PolicyCreated(uint indexed policyId, address indexed policyHolder);

    event ClaimSubmitted(uint indexed claimId, uint indexed policyId, uint amount, string reason);

    event ClaimApproved(uint indexed claimId, address indexed approver);

    modifier onlyOwner() {

        require(msg.sender == owner, "Only contract owner can perform this action");

        \_;

    }

    constructor() {

        owner = msg.sender;

        policyCounter = 0;

    }

    function createPolicy(address \_policyHolder, uint \_premium, uint \_coverage, uint \_expirationDate) public onlyOwner returns (uint) {

        policies[policyCounter] = Policy(policyCounter, \_policyHolder, \_premium, \_coverage, true, \_expirationDate, new uint[](0));

        emit PolicyCreated(policyCounter, \_policyHolder);

        policyCounter++;

        return policyCounter-1;

    }

    function submitClaim(uint \_policyId, uint \_amount, string memory \_reason) public {

        Policy storage myPolicy = policies[\_policyId];

        require(myPolicy.isActive == true, "Policy is not active");

        require(msg.sender == myPolicy.policyHolder, "Only policy holder can submit a claim");

        uint claimId = myPolicy.claimIds.length;

        myPolicy.claimIds.push(claimId);

        claims[claimId] = Claim(claimId, \_policyId, \_amount, \_reason, false, new address[](0));

        emit ClaimSubmitted(claimId, \_policyId, \_amount, \_reason);

    }

    function approveClaim(uint \_claimId) public onlyOwner {

        Claim storage myClaim = claims[\_claimId];

        require(myClaim.isApproved == false, "Claim has already been approved");

        myClaim.approvers.push(msg.sender);

        if (myClaim.approvers.length >= 2) {

            myClaim.isApproved = true;

            Policy storage myPolicy = policies[myClaim.policyId];

            myPolicy.isActive = false;

            myPolicy.claimIds = new uint[](0);

            emit ClaimApproved(\_claimId, msg.sender);

        }

    }

    function getPolicy(uint \_policyId) public view returns (address, uint, uint, bool, uint) {

        Policy storage myPolicy = policies[\_policyId];

        return (myPolicy.policyHolder, myPolicy.premium, myPolicy.coverage, myPolicy.isActive, myPolicy.expirationDate);

    }

    function getClaim(uint \_claimId) public view returns (uint, uint, uint, string memory, bool) {

        Claim storage myClaim = claims[\_claimId];

        return (myClaim.claimId, myClaim.policyId, myClaim.amount, myClaim.reason, myClaim.isApproved);

    }

}

INVENTORY MANAGEMENT

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract SupplyChain {

    address public manufacturer;

    address public retailer;

    uint public inventory;

    constructor() {

        manufacturer = msg.sender;

        inventory = 100;

    }

    function sellProduct(uint quantity) public {

        require(retailer == msg.sender, "Only retailer can sell products.");

        require(quantity <= inventory, "Not enough inventory.");

        inventory -= quantity;

    }

    function setRetailer(address \_retailer) public {

        require(manufacturer == msg.sender, "Only manufacturer can set retailer.");

        retailer = \_retailer;

    }

    function getInventory() public view returns (uint) {

        return inventory;

    }

}

contract Payment {

    mapping(address => uint) public balances;

    function deposit() public payable {

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

    }

    function withdraw(uint amount) public {

        require(balances[msg.sender] >= amount, "Not enough balance.");

        balances[msg.sender] -= amount;

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

    }

    function transfer(address recipient, uint amount) public {

        require(balances[msg.sender] >= amount, "Not enough balance.");

        balances[msg.sender] -= amount;

        balances[recipient] += amount;

    }

}

contract Marketplace {

    address public owner;

    Payment public paymentContract;

    mapping(address => bool) public retailers;

    mapping(bytes32 => uint) public prices;

    event ProductPurchased(address buyer, bytes32 product, uint quantity, uint price);

    constructor() {

        owner = msg.sender;

        paymentContract = new Payment();

    }

    function addRetailer(address retailer) public {

        require(msg.sender == owner, "Only owner can add retailer.");

        retailers[retailer] = true;

    }

    function addProduct(bytes32 product, uint price) public {

        require(retailers[msg.sender], "Only retailers can add products.");

        prices[product] = price;

    }

    function purchaseProduct(bytes32 product, uint quantity) public payable {

        require(prices[product] > 0, "Product does not exist.");

        require(msg.value >= prices[product] \* quantity, "Insufficient funds.");

        paymentContract.deposit{value: msg.value}();

        emit ProductPurchased(msg.sender, product, quantity, msg.value);

    }

}

IOT DATA

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract IoTData {

    struct SensorData {

        uint256 timestamp;

        uint256 value;

        string sensorType;

    }

    mapping(address => mapping(uint256 => SensorData)) private sensorData;

    function addSensorData(uint256 \_timestamp, uint256 \_value, string memory \_sensorType) public {

        SensorData memory data = SensorData(\_timestamp, \_value, \_sensorType);

        sensorData[msg.sender][\_timestamp] = data;

    }

    function getSensorData(address \_sensorAddress, uint256 \_timestamp) public view returns (uint256, string memory) {

        SensorData memory data = sensorData[\_sensorAddress][\_timestamp];

        return (data.value, data.sensorType);

    }

}

ONLINE PRODUCT

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract OnlineMarketplace {

    address public owner;

    struct Product {

        uint256 productId;

        string name;

        uint256 price;

        uint256 stock;

    }

    mapping(uint256 => Product) public products;

    uint256 public productCount;

    constructor() {

        owner = msg.sender;

    }

    modifier onlyOwner() {

        require(msg.sender == owner, "Only the owner can call this function");

        \_;

    }

    function addProduct(string memory \_name, uint256 \_price, uint256 \_stock) public onlyOwner {

        productCount++;

        products[productCount] = Product(productCount, \_name, \_price, \_stock);

    }

    function searchProduct(uint256 \_productId) public view returns (string memory, uint256, uint256) {

        require(\_productId <= productCount && \_productId > 0, "Invalid product ID");

        Product memory product = products[\_productId];

        return (product.name, product.price, product.stock);

    }

    function purchaseProduct(uint256 \_productId) public payable {

        require(\_productId <= productCount && \_productId > 0, "Invalid product ID");

        Product storage product = products[\_productId];

        require(product.stock > 0, "Product out of stock");

        require(msg.value >= product.price, "Insufficient funds");

        product.stock--;

        // Simulate shipping process

        // Transfer payment to the contract owner

        address payable ownerPayable = payable(owner);

        ownerPayable.transfer(product.price);

    }

}

PAYMENT OPERATION:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract PaymentProcessor {

    address public owner;

    uint256 public totalPayments;

    constructor() {

        owner = msg.sender;

    }

    event PaymentReceived(address indexed payer, uint256 amount);

    modifier onlyOwner() {

        require(msg.sender == owner, "Only the contract owner can call this function");

        \_;

    }

    function makePayment() public payable {

        require(msg.value > 0, "Payment amount must be greater than 0");

        totalPayments += msg.value;

        emit PaymentReceived(msg.sender, msg.value);

    }

    function withdrawFunds() public onlyOwner {

        require(totalPayments > 0, "No funds to withdraw");

        uint256 amountToWithdraw = totalPayments;

        totalPayments = 0;

        (bool success, ) = owner.call{value: amountToWithdraw}("");

        require(success, "Withdrawal failed");

    }

    function getContractBalance() public view returns (uint256) {

        return address(this).balance;

    }

}

PEERREVIEWING

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract PeerReview {

    struct Reviewer {

        bool isRegistered;

        bool hasReviewed;

    }

    struct Paper {

        uint id;

        address author;

        string title;

        string content;

        address[] reviewers;

        uint[] scores;

        uint numReviews;

        bool isAccepted;

    }

    mapping (address => Reviewer) public reviewers;

    mapping (uint => Paper) public papers;

    uint public paperCounter = 0;

    event PaperSubmitted(uint indexed paperId, address indexed author);

    event PaperReviewed(uint indexed paperId, address indexed reviewer, uint score);

    event PaperAccepted(uint indexed paperId);

    function registerAsReviewer() public {

        reviewers[msg.sender].isRegistered = true;

    }

    function submitPaper(string memory \_title, string memory \_content, address[] memory \_reviewers) public {

        require(\_reviewers.length >= 2, "Papers require at least two reviewers");

        paperCounter++;

        Paper storage paper = papers[paperCounter];

        paper.id = paperCounter;

        paper.author = msg.sender;

        paper.title = \_title;

        paper.content = \_content;

        paper.reviewers = \_reviewers;

        paper.scores = new uint[](\_reviewers.length);

        paper.numReviews = 0;

        paper.isAccepted = false;

        emit PaperSubmitted(paperCounter, msg.sender);

    }

    function reviewPaper(uint \_paperId, uint \_score) public {

        Paper storage paper = papers[\_paperId];

        require(reviewers[msg.sender].isRegistered, "You must be registered as a reviewer to review papers");

        require(!paper.hasReviewed[msg.sender], "You have already reviewed this paper");

        require(paper.numReviews < paper.reviewers.length, "All reviews for this paper have already been submitted");

        paper.scores[paper.numReviews] = \_score;

        paper.numReviews++;

        if (paper.numReviews == paper.reviewers.length) {

            uint totalScore = 0;

            for (uint i = 0; i < paper.scores.length; i++) {

                totalScore += paper.scores[i];

            }

            uint averageScore = totalScore / paper.scores.length;

            if (averageScore >= 70) {

                paper.isAccepted = true;

                emit PaperAccepted(\_paperId);

            }

        }

        paper.hasReviewed[msg.sender] = true;

        emit PaperReviewed(\_paperId, msg.sender, \_score);

    }

    function getPaper(uint \_paperId) public view returns (string memory, string memory, bool, address) {

        Paper storage paper = papers[\_paperId];

        return (paper.title, paper.content, paper.isAccepted, paper.author);

    }

}

SUPPLYCHAIN:

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract SupplyChain {

    struct Product {

        uint productId;

        string name;

        address owner;

        address[] history;

    }

    mapping(uint => Product) public products;

    uint public productCounter;

    event ProductCreated(uint indexed productId, string name, address indexed owner);

    event ProductTransferred(uint indexed productId, address indexed from, address indexed to);

    function createProduct(string memory \_name) public returns (uint) {

        products[productCounter] = Product(productCounter, \_name, msg.sender, new address[](0));

        emit ProductCreated(productCounter, \_name, msg.sender);

        productCounter++;

        return productCounter-1;

    }

    function transferProduct(uint \_productId, address \_to) public {

        Product storage myProduct = products[\_productId];

        require(msg.sender == myProduct.owner, "Only product owner can transfer the product");

        myProduct.owner = \_to;

        myProduct.history.push(\_to);

        emit ProductTransferred(\_productId, msg.sender, \_to);

    }

    function getProductHistory(uint \_productId) public view returns (address[] memory) {

        return products[\_productId].history;

    }

}

TRANSCRIPT:

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Verification {

    constructor() { owner = msg.sender; }

    uint16 public count\_Exporters =0;

    uint16 public count\_hashes=0;

    address public owner;

    struct  Record  {

        uint blockNumber;

        uint minetime;

        string info;

        string ipfs\_hash;

         }

    struct Exporter\_Record{

        uint blockNumber;

        string info;

         }

     mapping (bytes32  => Record) private docHashes;

     mapping (address => Exporter\_Record) private Exporters;

//---------------------------------------------------------------------------------------------------------//

    modifier onlyOwner() {

            if (msg.sender != owner) {

            revert("Caller is not the owner"); }\_; }

    modifier validAddress(address \_addr) {

            assert(\_addr != address(0)); \_; }

    modifier authorised\_Exporter(bytes32  \_doc){

         if (keccak256(abi.encodePacked((Exporters[msg.sender].info )))!= keccak256(abi.encodePacked((docHashes[\_doc].info))))

        revert("Caller is not  authorised to edit this document");

         \_; }

    modifier canAddHash(){

        require(Exporters[msg.sender].blockNumber!=0,"Caller not authorised to add documents");   \_; }

//---------------------------------------------------------------------------------------------------------//

    function add\_Exporter(address \_add,string calldata \_info) external

    onlyOwner(){

        assert(Exporters[\_add].blockNumber==0);

          Exporters[\_add].blockNumber = block.number;

          Exporters[\_add].info = \_info;

          ++count\_Exporters;

        }

    function delete\_Exporter(address \_add) external

    onlyOwner

    {

        assert(Exporters[\_add].blockNumber!=0);

        Exporters[\_add].blockNumber=0;

        Exporters[\_add].info="";

        --count\_Exporters;

        }

    function alter\_Exporter(address \_add,string calldata    \_newInfo) public

    onlyOwner()

     {

          assert(Exporters[\_add].blockNumber!=0);

             Exporters[\_add].info=\_newInfo; }

    function changeOwner(address \_newOwner) public

        onlyOwner  validAddress(\_newOwner)   {  owner = \_newOwner; }

        event addHash(address indexed \_exporter,string \_ipfsHash);

    function addDocHash (bytes32  hash,string calldata \_ipfs) public

      canAddHash

      {

            assert(docHashes[hash].blockNumber==0 && docHashes[hash].minetime==0);

            Record memory  newRecord =

            Record(block.number,block.timestamp,Exporters[msg.sender].info,\_ipfs);

            docHashes[hash] = newRecord;

            ++count\_hashes;

            emit addHash(msg.sender,\_ipfs);

      }

    function findDocHash (bytes32 \_hash)

    external  view  returns (uint,uint,string memory,string memory) {

        return (docHashes[\_hash].blockNumber,docHashes[\_hash].minetime,docHashes[\_hash].info,docHashes[\_hash].ipfs\_hash );

        }

    function deleteHash (bytes32 \_hash) public

    authorised\_Exporter(\_hash)

    canAddHash

    {

    assert(docHashes[\_hash].minetime!=0);

    docHashes[\_hash].blockNumber=0;

    docHashes[\_hash].minetime=0;

    --count\_hashes;

    }

    function  getExporterInfo(address \_add) external view returns(string memory){

        return (Exporters[\_add].info);

    }

}

RESTRICTED WITHDRAWL:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract WithdrawalRestrictions {

    address public owner;

    mapping(address => uint) public balances;

    mapping(address => uint) public securityLevels;

    enum SecurityLevel {

        LOW,

        MEDIUM,

        HIGH

    }

    modifier onlyOwner() {

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

        \_;

    }

    modifier hasSufficientSecurity(SecurityLevel requiredLevel) {

        require(securityLevels[msg.sender] >= uint(requiredLevel), "Insufficient security level for this operation.");

        \_;

    }

    constructor() {

        owner = msg.sender;

        securityLevels[owner] = uint(SecurityLevel.HIGH);

    }

    function setSecurityLevel(address account, SecurityLevel level) public onlyOwner {

        securityLevels[account] = uint(level);

    }

    function deposit() public payable {

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

    }

    function withdraw(uint amount) public hasSufficientSecurity(SecurityLevel.MEDIUM) {

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

        balances[msg.sender] -= amount;

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

    }

}

RENTAL:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract LeaseContract {

    address public owner;

    address public tenant;

    uint256 public rentAmount;

    uint256 public depositAmount;

    bool public isActive;

    event LeaseCreated(address indexed owner, address indexed tenant, uint256 rentAmount, uint256 depositAmount);

    event LeaseTerminated(address indexed owner, address indexed tenant);

    event DepositSubmitted(address indexed tenant, uint256 depositAmount);

    constructor(address \_owner, address \_tenant, uint256 \_rentAmount, uint256 \_depositAmount) {

        require(\_owner != address(0), "Owner address cannot be zero");

        require(\_tenant != address(0), "Tenant address cannot be zero");

        require(\_rentAmount > 0, "Rent amount must be greater than zero");

        require(\_depositAmount >= 0, "Deposit amount cannot be negative");

        owner = \_owner;

        tenant = \_tenant;

        rentAmount = \_rentAmount;

        depositAmount = \_depositAmount;

        isActive = true;

        emit LeaseCreated(owner, tenant, rentAmount, depositAmount);

    }

    // Function to terminate the lease

    function terminateLease() public {

        require(msg.sender == owner || msg.sender == tenant, "Only owner or tenant can terminate the lease");

        isActive = false;

        emit LeaseTerminated(owner, tenant);

    }

    // Function for the tenant to submit a deposit

    function submitDeposit(uint256 \_amount) public {

        require(msg.sender == tenant, "Only the tenant can submit a deposit");

        require(\_amount >= depositAmount, "Deposit amount should be at least the required deposit");

        // You can handle the deposit transfer logic here, for example:

        // require(address(this).balance >= \_amount, "Contract balance insufficient");

        // payable(owner).transfer(\_amount); // Transfer deposit to owner

        emit DepositSubmitted(tenant, \_amount);

    }

    // Function to get the balance of this lease contract

    function getBalance() public view returns (uint256) {

        return address(this).balance;

    }

}

contract LeaseManager {

    address public owner;

    mapping(uint256 => LeaseContract) public leases;

    uint256 public leaseCount;

    event LeaseContractCreated(uint256 indexed leaseId, address indexed owner, address indexed tenant, uint256 rentAmount, uint256 depositAmount);

    constructor() {

        owner = msg.sender;

    }

    modifier onlyOwner() {

        require(msg.sender == owner, "Only the owner can call this function");

        \_;

    }

    // Function to create a new lease contract

    function createLease(address \_tenant, uint256 \_rentAmount, uint256 \_depositAmount) public onlyOwner {

        require(\_tenant != address(0), "Tenant address cannot be zero");

        require(\_rentAmount > 0, "Rent amount must be greater than zero");

        require(\_depositAmount >= 0, "Deposit amount cannot be negative");

        LeaseContract newLease = new LeaseContract(owner, \_tenant, \_rentAmount, \_depositAmount);

        leases[leaseCount] = newLease;

        leaseCount++;

        emit LeaseContractCreated(leaseCount - 1, owner, \_tenant, \_rentAmount, \_depositAmount);

    }

    // Function to get the total number of leases

    function getLeaseCount() public view returns (uint256) {

        return leaseCount;

    }

    // Function to check the balance of the contract

    function getContractBalance() public view onlyOwner returns (uint256) {

        return address(this).balance;

    }

}