CSIT 6000Q - Blockchain and Smart Contracts Assignment 2 Solutions

November 20, 2024

Answer 1: Decentralized Charity Fund Allocation with Governance Voting

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

/// @title Decentralized Charity Fund with Governance Voting
/// @notice A smart contract for managing decentralized donations and funding allocations with voting power proportional to contributions.
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract DecentralizedCharityFund is ReentrancyGuard {
    struct FundingRequest {
        address payable projectAddress; // Address of the project
        uint256 requestedAmount; // Requested amount in Wei
        string projectDescription; // Description of the project
        uint256 votesReceived; // Total votes received
        bool isFinalized; // Indicates if the request has been finalized
    }

    address public owner; // Contract owner
    uint256 public totalVotingPower; // Total voting power of all donors
```

```
    FundingRequest[] public fundingRequests; // Array of funding requests
    mapping(address => uint256) public donorVotingPower; // Mapping of donor addresses to their voting power
    mapping(uint256 => mapping(address => bool)) public hasVoted; // Tracks if a donor has voted on a specific request

    event DonationReceived(address indexed donor, uint256 amount);
    event FundingRequestSubmitted(uint256 requestId, address indexed projectAddress, uint256 requestedAmount, string projectDescription);
    event VoteCast(uint256 requestId, address indexed voter, uint256 votingPower);
    event RequestFinalized(uint256 requestId, address indexed projectAddress, uint256 amountDisbursed);

    constructor() {
        owner = msg.sender;
    }

    /// @notice Allows users to donate Ether and gain voting power proportional to the amount donated
    function donate() external payable {
        require(msg.value > 0, "Donation must be greater than zero");
        donorVotingPower[msg.sender] += msg.value;
        totalVotingPower += msg.value;

        emit DonationReceived(msg.sender, msg.value);
    }

    /// @notice Allows anyone to submit a funding request
    /// @param projectAddress The address of the project requesting funds
    /// @param requestedAmount The amount of Ether requested
    /// @param projectDescription A brief description of the project
    function submitFundingRequest(
        address payable projectAddress,
        uint256 requestedAmount,
        string calldata projectDescription
    ) external {
        require(projectAddress != address(0), "Invalid
```

```
    project address");
51          require(requestedAmount > 0, "Requested amount must
    be greater than zero");
52
53          fundingRequests.push(FundingRequest({
54              projectAddress: projectAddress,
55              requestedAmount: requestedAmount,
56              projectDescription: projectDescription,
57              votesReceived: 0,
58              isFinalized: false
59          }));
60
61          emit FundingRequestSubmitted(fundingRequests.length -
     1, projectAddress, requestedAmount, projectDescription);
62      }
63
64      /// @notice Allows donors to vote on a specific funding
    request using their voting power
65      /// @param requestId The ID of the funding request to
    vote on
66      function voteOnRequest(uint256 requestId) external {
67          require(requestId < fundingRequests.length, "Invalid
    request ID");
68          require(donorVotingPower[msg.sender] > 0, "No voting
    power");
69          require(!hasVoted[requestId][msg.sender], "Already
    voted on this request");
70          require(!fundingRequests[requestId].isFinalized, "
    Request already finalized");
71
72          fundingRequests[requestId].votesReceived +=
    donorVotingPower[msg.sender];
73          hasVoted[requestId][msg.sender] = true;
74
75          emit VoteCast(requestId, msg.sender, donorVotingPower
    [msg.sender]);
76      }
77
78      /// @notice Finalizes a funding request if it has
    received enough votes
79      /// @param requestId The ID of the funding request to
    finalize
80      function finalizeRequest(uint256 requestId) external
    nonReentrant {
81          require(requestId < fundingRequests.length, "Invalid
```

```
request ID");
82          FundingRequest storage request = fundingRequests[
requestId];
83          require(!request.isFinalized, "Request already
finalized");
84          require(request.votesReceived > totalVotingPower / 2,
 "Not enough votes to approve request");
85          require(address(this).balance >= request.
requestedAmount, "Insufficient contract balance");
86
87          // **Checks-Effects-Interactions** pattern for
reentrancy safety
88          request.isFinalized = true; // Mark the request as
finalized before interaction
89
90          // Transfer funds using call() for gas efficiency and
 error handling
91          (bool success, ) = request.projectAddress.call{value:
 request.requestedAmount}("");
92          require(success, "Transfer failed");
93
94          emit RequestFinalized(requestId, request.
projectAddress, request.requestedAmount);
95      }
96
97      /// @notice Retrieves the funding history of all
finalized requests
98      /// @return projectAddresses The addresses of funded
projects
99      /// @return amounts The amounts disbursed to each project
100     /// @return descriptions The descriptions of funded
projects
101     function getFundingHistory()
102         external
103         view
104         returns (
105             address[] memory projectAddresses,
106             uint256[] memory amounts,
107             string[] memory descriptions
108         )
109     {
110         uint256 fundedCount = 0;
111
112         // Count the number of funded projects
113         for (uint256 i = 0; i < fundingRequests.length; i++)
```

```
    {
114             if (fundingRequests[i].isFinalized) {
115                 fundedCount++;
116             }
117         }
118 
119         projectAddresses = new address[](fundedCount);
120         amounts = new uint256[](fundedCount);
121         descriptions = new string[](fundedCount);
122 
123         uint256 index = 0;
124         for (uint256 i = 0; i < fundingRequests.length; i++)
    {
125             if (fundingRequests[i].isFinalized) {
126                 projectAddresses[index] = fundingRequests[i].
    projectAddress;
127                 amounts[index] = fundingRequests[i].
    requestedAmount;
128                 descriptions[index] = fundingRequests[i].
    projectDescription;
129                 index++;
130             }
131         }
132     }
133 
134     /// @notice Allows the owner to withdraw any remaining
    balance in case of emergencies
135     function emergencyWithdraw() external {
136         require(msg.sender == owner, "Only the contract owner
     can withdraw funds");
137         payable(owner).transfer(address(this).balance);
138     }
139 }
```

Listing 1: Decentralized Charity Fund Allocation with Governance Voting

Answer 2: Decentralized Fan Engagement and Reward System

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.26;
3 
4 import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract FanToken is ERC20, Ownable {
    constructor(address initialOwner) ERC20("FanToken", "FAN") Ownable(initialOwner) {}

    function mint(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) external onlyOwner {
        _burn(from, amount);
    }
}

contract FanBadge is ERC721, Ownable {
    uint256 public nextTokenId;
    mapping(uint256 => string) public badgeNames;

    constructor(address initialOwner) ERC721("FanBadge", "FBG") Ownable(initialOwner) {}

    function mint(address to, string memory badgeName) external onlyOwner {
        _mint(to, nextTokenId);
        badgeNames[nextTokenId] = badgeName; // Store the badge name
        nextTokenId++;
    }

    function getBadgeName(uint256 tokenId) external view returns (string memory) {
        return badgeNames[tokenId];
    }
}

contract FanEngagementSystem is Ownable {
    FanToken public fanToken;
    FanBadge public fanBadge;

    enum LoyaltyTier { Bronze, Silver, Gold }
    struct Fan {
```

```
        uint256 tokenBalance;
        LoyaltyTier loyaltyTier;
        string[] rewardHistory;
        mapping(string => bool) badgesOwned;
    }

    struct Proposal {
        string description;
        uint256 voteCount;
        bool finalized;
        mapping(address => bool) hasVoted;
    }

    mapping(address => Fan) public fans;
    mapping(uint256 => Proposal) public proposals;
    uint256 public proposalCounter;

    uint256 public constant BRONZE_THRESHOLD = 100;
    uint256 public constant SILVER_THRESHOLD = 500;
    uint256 public constant GOLD_THRESHOLD = 1000;

    event TokensEarned(address indexed fan, uint256 amount,
string activityType);
    event TokensTransferred(address indexed from, address
indexed to, uint256 amount);
    event TokensRedeemed(address indexed fan, uint256 amount,
     string rewardType);
    event NFTBadgeMinted(address indexed fan, string
badgeName);
    event ProposalSubmitted(uint256 indexed proposalId,
string description);
    event VotedOnProposal(uint256 indexed proposalId, address
     voter);

    constructor(address _fanToken, address _fanBadge, address
     initialOwner) Ownable(initialOwner) {
        fanToken = FanToken(_fanToken);
        fanBadge = FanBadge(_fanBadge);
    }

    // Earn tokens for activities
    function earnTokens(address fan, uint256 amount, string
memory activityType) external onlyOwner {
        require(amount > 0, "Amount must be greater than zero
.");
```

```
79          fanToken.mint(fan, amount);
80          fans[fan].tokenBalance += amount;
81          updateLoyaltyTier(fan);
82          emit TokensEarned(fan, amount, activityType);
83      }
84
85      // Transfer tokens between fans
86      function transferTokens(address to, uint256 amount)
    external {
87          require(fans[msg.sender].tokenBalance >= amount, "
    Insufficient token balance.");
88          require(to != address(0), "Cannot transfer to zero
    address.");
89
90          fanToken.approve(address(this), amount);
91          fanToken.transferFrom(msg.sender, to, amount);
92
93          fans[msg.sender].tokenBalance -= amount;
94          fans[to].tokenBalance += amount;
95
96          updateLoyaltyTier(msg.sender);
97          updateLoyaltyTier(to);
98
99          emit TokensTransferred(msg.sender, to, amount);
100     }
101
102     // Redeem tokens for rewards
103     function redeemTokens(uint256 amount, string memory
    rewardType) external {
104         require(fans[msg.sender].tokenBalance >= amount, "
    Insufficient token balance.");
105         require(bytes(rewardType).length > 0, "Reward type
    cannot be empty.");
106
107         fanToken.burn(msg.sender, amount);
108         fans[msg.sender].tokenBalance -= amount;
109         fans[msg.sender].rewardHistory.push(rewardType);
110
111         updateLoyaltyTier(msg.sender);
112         emit TokensRedeemed(msg.sender, amount, rewardType);
113     }
114
115     // Mint NFT badges
116     function mintNFTBadge(address fan, string memory
    badgeName) external onlyOwner {
```

```
        require(!fans[fan].badgesOwned[badgeName], "Badge already owned.");
        require(bytes(badgeName).length > 0, "Badge name cannot be empty.");

        fanBadge.mint(fan, badgeName);
        fans[fan].badgesOwned[badgeName] = true;
        emit NFTBadgeMinted(fan, badgeName);
    }

    // Submit a proposal
    function submitProposal(string memory proposalDescription) external {
        require(fans[msg.sender].tokenBalance > 0, "Must hold tokens to submit proposal.");
        require(bytes(proposalDescription).length > 0, "Proposal description cannot be empty.");

        Proposal storage newProposal = proposals[proposalCounter];
        newProposal.description = proposalDescription;
        newProposal.finalized = false;

        emit ProposalSubmitted(proposalCounter, proposalDescription);
        proposalCounter++;
    }

    // Vote on a proposal
    function voteOnProposal(uint256 proposalId) external {
        require(proposalId < proposalCounter, "Invalid proposal ID.");
        require(!proposals[proposalId].hasVoted[msg.sender], "Already voted on this proposal.");
        require(fans[msg.sender].tokenBalance > 0, "Must hold tokens to vote.");

        Proposal storage proposal = proposals[proposalId];
        proposal.voteCount += fans[msg.sender].tokenBalance;
        proposal.hasVoted[msg.sender] = true;

        emit VotedOnProposal(proposalId, msg.sender);
    }

    // Get fan loyalty tier
```

```
    function getFanLoyaltyTier(address fan) external view returns (string memory) {
        LoyaltyTier tier = fans[fan].loyaltyTier;
        if (tier == LoyaltyTier.Gold) return "Gold";
        if (tier == LoyaltyTier.Silver) return "Silver";
        return "Bronze";
    }

    // Get fan reward history
    function getRewardHistory(address fan) external view returns (string[] memory) {
        return fans[fan].rewardHistory;
    }

    // Private function to update loyalty tier
    function updateLoyaltyTier(address fan) private {
        uint256 balance = fans[fan].tokenBalance;
        if (balance >= GOLD_THRESHOLD) {
            fans[fan].loyaltyTier = LoyaltyTier.Gold;
        } else if (balance >= SILVER_THRESHOLD) {
            fans[fan].loyaltyTier = LoyaltyTier.Silver;
        } else if (balance >= BRONZE_THRESHOLD) {
            fans[fan].loyaltyTier = LoyaltyTier.Bronze;
        }
    }
}
```

Listing 2: Decentralized Fan Engagement and Reward System

Answer 3: Decentralized Rental Agreement Management

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

/**
 * @title Rental Agreement Management
 * @dev A blockchain-based system for managing rental agreements, allowing landlords to create agreements,
 * tenants to pay rent, and landlords to terminate agreements with comprehensive event logging.
 */
contract RentalAgreementManagement {
```

```
    uint256 public agreementCounter;
    bool private locked;

    struct Agreement {
        address landlord;
        address tenant;
        uint256 rentAmount;
        uint256 duration; // in days
        uint256 startDate;
        uint256 totalPaid;
        bool isActive;
    }

    mapping(uint256 => Agreement) public agreements;
    mapping(uint256 => uint256[]) public paymentTimestamps;

    event AgreementCreated(
        uint256 indexed agreementId,
        address indexed landlord,
        address indexed tenant,
        uint256 rentAmount,
        uint256 duration
    );
    event RentPaid(uint256 indexed agreementId, address
    indexed tenant, uint256 amount, uint256 timestamp);
    event AgreementTerminated(uint256 indexed agreementId,
    address indexed landlord);

    constructor() {
        agreementCounter = 0;
    }

    modifier nonReentrant() {
        require(!locked, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }

    /**
     * @dev Create a new rental agreement.
     * @param tenant The address of the tenant.
     * @param rentAmount The rent amount for the agreement.
     * @param duration The duration of the agreement in days.
     */
```

```
53      function createAgreement(address tenant, uint256
    rentAmount, uint256 duration) external {
54          require(tenant != address(0), "Invalid tenant address
    .");
55          require(rentAmount > 0, "Rent amount must be greater
    than zero.");
56          require(duration > 0, "Duration must be greater than
    zero.");
57
58          agreements[agreementCounter] = Agreement({
59              landlord: msg.sender,
60              tenant: tenant,
61              rentAmount: rentAmount,
62              duration: duration,
63              startDate: block.timestamp,
64              totalPaid: 0,
65              isActive: true
66          });
67
68          emit AgreementCreated(agreementCounter, msg.sender,
    tenant, rentAmount, duration);
69          agreementCounter++;
70      }
71
72      /**
73       * @dev Pay rent for a specific agreement.
74       * @param agreementId The ID of the agreement.
75       */
76      function payRent(uint256 agreementId) external payable
    nonReentrant {
77          Agreement storage agreement = agreements[agreementId
    ];
78          require(agreement.isActive, "Agreement is not active.
    ");
79          require(msg.sender == agreement.tenant, "Only the
    tenant can pay rent.");
80          require(msg.value == agreement.rentAmount, "Incorrect
     rent amount.");
81          require(block.timestamp <= agreement.startDate +
    agreement.duration * 1 days, "Agreement has expired.");
82
83          agreement.totalPaid += msg.value;
84          paymentTimestamps[agreementId].push(block.timestamp);
85
86          // Using `call` for safer Ether transfer
```

```
87          (bool success, ) = agreement.landlord.call{value: msg
    .value}("");
88          require(success, "Rent payment failed.");
89
90          emit RentPaid(agreementId, msg.sender, msg.value,
    block.timestamp);
91      }
92
93      /**
94       * @dev Terminate a rental agreement.
95       * @param agreementId The ID of the agreement.
96       */
97      function terminateAgreement(uint256 agreementId) external
        {
98          Agreement storage agreement = agreements[agreementId
    ];
99          require(msg.sender == agreement.landlord, "Only the
    landlord can terminate the agreement.");
100         require(agreement.isActive, "Agreement is already
    terminated.");
101         require(block.timestamp >= agreement.startDate +
    agreement.duration * 1 days, "Cannot terminate before
    lease ends.");
102
103         agreement.isActive = false;
104         emit AgreementTerminated(agreementId, msg.sender);
105     }
106
107     /**
108      * @dev Get the status of a rental agreement.
109      * @param agreementId The ID of the agreement.
110      * @return The status of the agreement ("Active", "
    Expired", or "Terminated").
111      */
112     function getAgreementStatus(uint256 agreementId) external
        view returns (string memory) {
113         Agreement storage agreement = agreements[agreementId
    ];
114         if (!agreement.isActive) {
115             return "Terminated";
116         } else if (block.timestamp > agreement.startDate +
    agreement.duration * 1 days) {
117             return "Expired";
118         } else {
119             return "Active";
```

```
        }
    }

    /**
     * @dev Get the payment history of a rental agreement.
     * @param agreementId The ID of the agreement.
     * @return An array of timestamps indicating when payments were made.
     */
    function getPaymentHistory(uint256 agreementId) external view returns (uint256[] memory) {
        return paymentTimestamps[agreementId];
    }
}
```

Listing 3: Rental Agreement Management

Answer 4: Decentralized Auction House

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

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract DecentralizedAuctionHouse is ReentrancyGuard {
    struct Auction {
        uint256 id;
        address payable artist;
        string itemName;
        uint256 reservePrice;
        uint256 highestBid;
        address payable highestBidder;
        uint256 endTime;
        bool finalized;
    }

    uint256 public auctionCount;
    mapping(uint256 => Auction) public auctions;
    mapping(uint256 => mapping(address => uint256)) public bids;

    uint256 public constant LOCK_PERIOD = 300; // Lock period for bid withdrawal
```

```
    uint256 public constant EXTENSION_TIME = 300; // Time extension for late bids

    event AuctionCreated(
        uint256 indexed auctionId,
        address indexed artist,
        string itemName,
        uint256 reservePrice,
        uint256 endTime
    );
    event BidPlaced(
        uint256 indexed auctionId,
        address indexed bidder,
        uint256 amount
    );
    event BidWithdrawn(
        uint256 indexed auctionId,
        address indexed bidder,
        uint256 amount
    );
    event AuctionFinalized(
        uint256 indexed auctionId,
        address indexed artist,
        address indexed winner,
        uint256 amount
    );

    constructor() {
        auctionCount = 0;
    }

    function createAuction(
        string memory itemName,
        uint256 reservePrice,
        uint256 auctionDuration
    ) external {
        require(reservePrice > 0, "Reserve price must be greater than zero");
        require(auctionDuration > 0, "Auction duration must be greater than zero");

        auctionCount++;
        uint256 endTime = block.timestamp + auctionDuration;

        auctions[auctionCount] = Auction({
```

```
            id: auctionCount,
            artist: payable(msg.sender),
            itemName: itemName,
            reservePrice: reservePrice,
            highestBid: 0,
            highestBidder: payable(address(0)),
            endTime: endTime,
            finalized: false
        });

        emit AuctionCreated(auctionCount, msg.sender,
    itemName, reservePrice, endTime);
    }

    function placeBid(uint256 auctionId) external payable
    nonReentrant {
        Auction storage auction = auctions[auctionId];
        require(block.timestamp < auction.endTime, "Auction
    has ended");
        require(msg.value > auction.highestBid, "Bid must be
    higher than current highest bid");
        require(msg.value >= auction.reservePrice, "Bid must
    meet reserve price");

        // Refund the previous highest bidder
        if (auction.highestBidder != address(0)) {
            bids[auctionId][auction.highestBidder] += auction
    .highestBid;
        }

        // Extend auction time if bid is placed close to the
    end
        if (block.timestamp + EXTENSION_TIME >= auction.
    endTime) {
            auction.endTime += EXTENSION_TIME;
        }

        // Update auction state
        auction.highestBid = msg.value;
        auction.highestBidder = payable(msg.sender);

        emit BidPlaced(auctionId, msg.sender, msg.value);
    }

    function withdrawBid(uint256 auctionId) external
```

```
    nonReentrant {
102         Auction storage auction = auctions[auctionId];
103         uint256 amount = bids[auctionId][msg.sender];
104 
105         require(block.timestamp < auction.endTime, "Auction
    has ended");
106         require(msg.sender != auction.highestBidder, "Highest
     bidder cannot withdraw");
107         require(amount > 0, "No bid to withdraw");
108 
109         // Reset the bid amount and transfer funds back
110         bids[auctionId][msg.sender] = 0;
111         (bool success, ) = payable(msg.sender).call{value:
    amount}("");
112         require(success, "Transfer failed");
113 
114         emit BidWithdrawn(auctionId, msg.sender, amount);
115     }
116 
117     function finalizeAuction(uint256 auctionId) external
    nonReentrant {
118         Auction storage auction = auctions[auctionId];
119 
120         require(block.timestamp >= auction.endTime, "Auction
    has not ended yet");
121         require(!auction.finalized, "Auction already
    finalized");
122         require(msg.sender == auction.artist, "Only the
    artist can finalize");
123 
124         auction.finalized = true;
125 
126         if (auction.highestBid >= auction.reservePrice) {
127             // Transfer the winning bid to the artist
128             (bool success, ) = auction.artist.call{value:
    auction.highestBid}("");
129             require(success, "Transfer to artist failed");
130 
131             emit AuctionFinalized(
132                 auctionId,
133                 auction.artist,
134                 auction.highestBidder,
135                 auction.highestBid
136             );
137         } else {
```

```
            // Handle reserve price not met
            if (auction.highestBidder != address(0)) {
                bids[auctionId][auction.highestBidder] += auction.highestBid;
            }

            emit AuctionFinalized(auctionId, auction.artist, address(0), 0);
        }
    }

    function getAuctionDetails(uint256 auctionId)
        external
        view
        returns (
            string memory,
            uint256,
            uint256,
            uint256,
            address,
            bool
        )
    {
        Auction storage auction = auctions[auctionId];
        return (
            auction.itemName,
            auction.reservePrice,
            auction.highestBid,
            auction.endTime,
            auction.highestBidder,
            auction.finalized
        );
    }
}
```

Listing 4: Decentralized Auction House