```
# Attribution:

Most of the code behind the functions and the CloneFactory in the CSC590DAOv7 are
taken and/or refactored from the Molochv2.1 framework
as expressed in the project's presentation and understood between group members and
instructor. This is not meant to claim them as original
work as far as attribution is concerned. The purpose of the exercise is and always has
been to write a DAO contract using a framework in
order to better understand the structure and function of DAO contracts. The
deliverable of this project is a mini-DAO contract which is
refactored largely from the framework listed in order to demonstrate understanding of
the variable components of DAOs that enable their function.
By including the FlatMolochFramework.sol and MolochREADME.md documents in this repo,
the author's intention is to provide as much transparency
as possible in the inclusion and reliance on the framework utilized.
```

```
/*
The MIT License (MIT)
Copyright (c) 2018 Murray Software, LLC.
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

pragma solidity 0.5.3;

contract CloneFactory { // implementation of eip-1167 - see
https://eips.ethereum.org/EIPS/eip-1167
   function createClone(address target) internal returns (address result) {
```

```
        bytes20 targetBytes = bytes20(target);
        assembly {
            let clone := mload(0x40)
            mstore(clone,
0x3d602d80600a3d3981f3363d3d373d3d3d363d730000000000000000000000000000)
            mstore(add(clone, 0x14), targetBytes)
            mstore(add(clone, 0x28),
0x5af43d82803e903d91602b57fd5bf3000000000000000000000000000000000000)
            result := create(0, clone, 0x37)
        }
    }
}
```

```
pragma solidity 0.5.3;

import "./CSC590DAOPROJECT.sol";
import "./CloneFactory.sol";

contract CSC590DAOSummoner is CloneFactory {

    address public template;
    mapping (address => bool) public daos;
    uint daoIdx = 0;
    CSC590DAOSummoner private csc590dao; // stating the contract

    constructor(address _template) public {
        template = _template;
    }

    event SummonComplete(address indexed csc590dao, address[] summoner, address[]
tokens, uint256 summoningTime, uint256 periodDuration, uint256 votingPeriodLength,
uint256 gracePeriodLength, uint256 proposalDeposit, uint256 dilutionBound, uint256
processingReward, uint256[] summonerShares);
    event Register(uint daoIdx, address csc590dao, string title, string http, uint
version);

    function summonCsc590dao(
        address[] memory _summoner,
```

```
        address[] memory _approvedTokens,
        uint256 _periodDuration,
        uint256 _votingPeriodLength,
        uint256 _gracePeriodLength,
        uint256 _proposalDeposit,
        uint256 _dilutionBound,
        uint256 _processingReward,
        uint256[] memory _summonerShares
    ) public returns (address) {
        Csc590dao baal = Csc590dao(createClone(template));

        baal.init(
            _summoner,
            _approvedTokens,
            _periodDuration,
            _votingPeriodLength,
            _gracePeriodLength,
            _proposalDeposit,
            _dilutionBound,
            _processingReward,
            _summonerShares
        );

        emit SummonComplete(address(baal), _summoner, _approvedTokens, now,
_periodDuration, _votingPeriodLength, _gracePeriodLength, _proposalDeposit,
_dilutionBound, _processingReward, _summonerShares);

        return address(baal);
    }

    //Registering function

    function registerDao(
        address _daoAdress,
        string memory _daoTitle,
        string memory _http,
        uint _version
      ) public returns (bool) {

      csc590dao = Csc590dao(_daoAdress);
      (,,,bool exists,,) = csc590dao.members(msg.sender);
```

```
    require(exists == true, "must be a member");
    require(daos[_daoAdress] == false, "dao metadata already registered");

    daos[_daoAdress] = true;

    daoIdx = daoIdx + 1;
    emit Register(daoIdx, _daoAdress, _daoTitle, _http, _version);
    return true;

  }
}
```

```
pragma solidity 0.5.3;

import "./oz/IERC20.sol";
import "./oz/SafeMath.sol";
import "./oz/ReentrancyGuard.sol";

contract CSC590DAOPROJECT is ReentrancyGuard {
  using SafeMath for uint256;


  //Global constants

  uint256 public periodDuration; // default = 17280 = 4.8 hours in seconds (5 periods
per day)
  uint256 public votingPeriodLength; // default = 35 periods (7 days)
  uint256 public gracePeriodLength; // default = 35 periods (7 days)
  uint256 public proposalDeposit; // default = 10 ETH (~$1,000 worth of ETH at
contract deployment)
  uint256 public dilutionBound; // default = 3 - maximum multiplier a YES voter will
be obligated to pay in case of mass DAO withdrawal
  uint256 public processingReward; // default = 0.1 - amount of ETH to give to
whoever processes a proposal
  uint256 public summoningTime; // needed to determine the current period
  bool private initialized; // internally tracks deployment under eip-1167 proxy
pattern
```

```
    address public depositToken; // deposit token contract reference; default = wETH


    // Hard-coded limits
    // These numbers are all arbitrary, but they'e small enough to avoid overflows when doing calculations and big enough to not limit reasonable use cases.

    uint256 constant MAX_VOTING_PERIOD_LENGTH = 10**18; // maximum length of voting period
    uint256 constant MAX_GRACE_PERIOD_LENGTH = 10**18; // maximum length of grace period
    uint256 constant MAX_DILUTION_BOUND = 10**18; // maximum dilution bound
    uint256 constant MAX_NUMBER_OF_SHARES_AND_LOOT = 10**18; // maximum number of shares that can be minted
    uint256 constant MAX_TOKEN_WHITELIST_COUNT = 400; // maximum number of whitelisted tokens
    uint256 constant MAX_TOKEN_GUILDBANK_COUNT = 200; // maximum number of tokens with non-zero balance in guildbank


    // Internal stuff

    //Below is the voting mechanism, with "Null" as the default value which the system counts as abstention
    enum Vote {
        Null,
        Yes,
        No
    }

    struct Member { //Member properties used in proposals, accounting and keeping tabs on individuals

        address delegateKey; // the key responsible for submitting proposals and voting - defaults to member address unless updated
        uint256 shares; // the # of voting shares assigned to this member
        uint256 loot; // the loot amount available to this member (combined with shares on ragequit)
        bool exists; // always true once a member has been created
        uint256 highestIndexYesVote; // highest proposal index # on which the member voted YES
```

```
        uint256 jailed; // set to proposalIndex of a passing guild kick proposal for
this member, prevents voting on and sponsoring proposals

    }

    struct Proposal { //Legal stuff as far as the structure of proposals, using
uint256

        address applicant; // the applicant who wishes to become a member - this key
will be used for withdrawals (doubles as guild kick target for gkick proposals)
        address proposer; // the account that submitted the proposal (can be
non-member)
        address sponsor; // the member that sponsored the proposal (moving it into the
queue)
        uint256 sharesRequested; // the # of shares the applicant is requesting
        uint256 lootRequested; // the amount of loot the applicant is requesting
        uint256 tributeOffered; // amount of tokens offered as tribute
        address tributeToken; // tribute token contract reference
        uint256 paymentRequested; // amount of tokens requested as payment
        address paymentToken; // payment token contract reference
        uint256 startingPeriod; // the period in which voting can start for this
proposal
        uint256 yesVotes; // the total number of YES votes for this proposal
        uint256 noVotes; // the total number of NO votes for this proposal
        bool[6] flags; // [sponsored, processed, didPass, cancelled, whitelist,
guildkick]
        string details; // proposal details - could be IPFS hash, plaintext, or JSON
        uint256 maxTotalSharesAndLootAtYesVote; // the maximum # of total shares
encountered at a yes vote on this proposal
        mapping(address => Vote) votesByMember; // the votes on this proposal by each
member
    }

        //Events that can happen in the running of the DAO, like withdrawing from the
DAO, kicking members, withdrawing tokens, voting, etc.

        event SummonComplete(address indexed summoner, address[] tokens, uint256
summoningTime, uint256 periodDuration, uint256 votingPeriodLength, uint256
gracePeriodLength, uint256 proposalDeposit, uint256 dilutionBound, uint256
processingReward);
        event SubmitProposal(address indexed applicant, uint256 sharesRequested,
uint256 lootRequested, uint256 tributeOffered, address tributeToken, uint256
```

```
paymentRequested, address paymentToken, string details, bool[6] flags, uint256
proposalId, address indexed delegateKey, address indexed memberAddress);
       event SponsorProposal(address indexed delegateKey, address indexed
memberAddress, uint256 proposalId, uint256 proposalIndex, uint256 startingPeriod);
       event SubmitVote(uint256 proposalId, uint256 indexed proposalIndex, address
indexed delegateKey, address indexed memberAddress, uint8 uintVote);
       event ProcessProposal(uint256 indexed proposalIndex, uint256 indexed
proposalId, bool didPass);
       event ProcessWhitelistProposal(uint256 indexed proposalIndex, uint256 indexed
proposalId, bool didPass);
       event ProcessGuildKickProposal(uint256 indexed proposalIndex, uint256 indexed
proposalId, bool didPass);
       event DAOWithdraw(address indexed memberAddress, uint256 sharesToBurn, uint256
lootToBurn);
       event TokensCollected(address indexed token, uint256 amountToCollect);
       event CancelProposal(uint256 indexed proposalId, address applicantAddress);
       event UpdateDelegateKey(address indexed memberAddress, address newDelegateKey);
       event Withdraw(address indexed memberAddress, address token, uint256 amount);

       //Functions

       //Proposal function

       function _submitProposal(
       address applicant,
       uint256 sharesRequested,
       uint256 lootRequested,
       uint256 tributeOffered,
       address tributeToken,
       uint256 paymentRequested,
       address paymentToken,
       string memory details,
       bool[6] memory flags
   ) internal {
       Proposal memory proposal = Proposal({
           applicant : applicant,
           proposer : msg.sender,
           sponsor : address(0),
           sharesRequested : sharesRequested,
           lootRequested : lootRequested,
           tributeOffered : tributeOffered,
           tributeToken : tributeToken,
```

```
            paymentRequested : paymentRequested,
            paymentToken : paymentToken,
            startingPeriod : 0,
            yesVotes : 0,
            noVotes : 0,
            flags : flags,
            details : details,
            maxTotalSharesAndLootAtYesVote : 0
        });

        proposals[proposalCount] = proposal;
        address memberAddress = memberAddressByDelegateKey[msg.sender];
        // NOTE: argument order matters, avoid stack too deep
        emit SubmitProposal(applicant, sharesRequested, lootRequested, tributeOffered,
tributeToken, paymentRequested, paymentToken, details, flags, proposalCount,
msg.sender, memberAddress);
        proposalCount += 1;
    }

    function sponsorProposal(uint256 proposalId) public nonReentrant onlyDelegate {
        // collect proposal deposit from sponsor and store it in the Moloch until the
proposal is processed
        require(IERC20(depositToken).transferFrom(msg.sender, address(this),
proposalDeposit), "proposal deposit token transfer failed");
        unsafeAddToBalance(ESCROW, depositToken, proposalDeposit);

        Proposal storage proposal = proposals[proposalId];

        require(proposal.proposer != address(0), 'proposal must have been proposed');
        require(!proposal.flags[0], "proposal has already been sponsored");
        require(!proposal.flags[3], "proposal has been cancelled");
        require(members[proposal.applicant].jailed == 0, "proposal applicant must not
be jailed");

        if (proposal.tributeOffered > 0 &&
userTokenBalances[GUILD][proposal.tributeToken] == 0) {
            require(totalGuildBankTokens < MAX_TOKEN_GUILDBANK_COUNT, 'cannot sponsor
more tribute proposals for new tokens - guildbank is full');
        }

        // whitelist proposal
        if (proposal.flags[4]) {
```

```
            require(!tokenWhitelist[address(proposal.tributeToken)], "cannot already
have whitelisted the token");
            require(!proposedToWhitelist[address(proposal.tributeToken)], 'already
proposed to whitelist');
            require(approvedTokens.length < MAX_TOKEN_WHITELIST_COUNT, "cannot sponsor
more whitelist proposals");
            proposedToWhitelist[address(proposal.tributeToken)] = true;

        // guild kick proposal
        } else if (proposal.flags[5]) {
            require(!proposedToKick[proposal.applicant], 'already proposed to kick');
            proposedToKick[proposal.applicant] = true;
        }

        // compute startingPeriod for proposal
        uint256 startingPeriod = max(
            getCurrentPeriod(),
            proposalQueue.length == 0 ? 0 :
proposals[proposalQueue[proposalQueue.length.sub(1)]].startingPeriod
        ).add(1);

        proposal.startingPeriod = startingPeriod;

        address memberAddress = memberAddressByDelegateKey[msg.sender];
        proposal.sponsor = memberAddress;

        proposal.flags[0] = true; // sponsored

        // append proposal to the queue
        proposalQueue.push(proposalId);

        emit SponsorProposal(msg.sender, memberAddress, proposalId,
proposalQueue.length.sub(1), startingPeriod);
    }

    // NOTE: In MolochV2 proposalIndex !== proposalId
    function submitVote(uint256 proposalIndex, uint8 uintVote) public nonReentrant
onlyDelegate {
        address memberAddress = memberAddressByDelegateKey[msg.sender];
        Member storage member = members[memberAddress];

        require(proposalIndex < proposalQueue.length, "proposal does not exist");
```

```
        Proposal storage proposal = proposals[proposalQueue[proposalIndex]];

        require(uintVote < 3, "must be less than 3");
        Vote vote = Vote(uintVote);

        require(getCurrentPeriod() >= proposal.startingPeriod, "voting period has not
started");
        require(!hasVotingPeriodExpired(proposal.startingPeriod), "proposal voting
period has expired");
        require(proposal.votesByMember[memberAddress] == Vote.Null, "member has already
voted");
        require(vote == Vote.Yes || vote == Vote.No, "vote must be either Yes or No");

        proposal.votesByMember[memberAddress] = vote;

        if (vote == Vote.Yes) {
            proposal.yesVotes = proposal.yesVotes.add(member.shares);

            // set highest index (latest) yes vote - must be processed for member to
ragequit
            if (proposalIndex > member.highestIndexYesVote) {
                member.highestIndexYesVote = proposalIndex;
            }

            // set maximum of total shares encountered at a yes vote - used to bound
dilution for yes voters
            if (totalShares.add(totalLoot) > proposal.maxTotalSharesAndLootAtYesVote) {
                proposal.maxTotalSharesAndLootAtYesVote = totalShares.add(totalLoot);
            }

        } else if (vote == Vote.No) {
            proposal.noVotes = proposal.noVotes.add(member.shares);
        }

        // NOTE: subgraph indexes by proposalId not proposalIndex since proposalIndex
isn't set untill it's been sponsored but proposal is created on submission
        emit SubmitVote(proposalQueue[proposalIndex], proposalIndex, msg.sender,
memberAddress, uintVote);
    }

    function processProposal(uint256 proposalIndex) public nonReentrant {
        _validateProposalForProcessing(proposalIndex);
```

```
        uint256 proposalId = proposalQueue[proposalIndex];
        Proposal storage proposal = proposals[proposalId];

        require(!proposal.flags[4] && !proposal.flags[5], "must be a standard
proposal");

        proposal.flags[1] = true; // processed

        bool didPass = _didPass(proposalIndex);

        // Make the proposal fail if the new total number of shares and loot exceeds
the limit
        if
(totalShares.add(totalLoot).add(proposal.sharesRequested).add(proposal.lootRequested)
> MAX_NUMBER_OF_SHARES_AND_LOOT) {
            didPass = false;
        }

        // Make the proposal fail if it is requesting more tokens as payment than the
available guild bank balance
        if (proposal.paymentRequested >
userTokenBalances[GUILD][proposal.paymentToken]) {
            didPass = false;
        }

        // Make the proposal fail if it would result in too many tokens with non-zero
balance in guild bank
        if (proposal.tributeOffered > 0 &&
userTokenBalances[GUILD][proposal.tributeToken] == 0 && totalGuildBankTokens >=
MAX_TOKEN_GUILDBANK_COUNT) {
           didPass = false;
        }

        // PROPOSAL PASSED
        if (didPass) {
            proposal.flags[2] = true; // didPass

            // if the applicant is already a member, add to their existing shares &
loot
            if (members[proposal.applicant].exists) {
```

```
                members[proposal.applicant].shares =
members[proposal.applicant].shares.add(proposal.sharesRequested);
                members[proposal.applicant].loot =
members[proposal.applicant].loot.add(proposal.lootRequested);

            // the applicant is a new member, create a new record for them
            } else {
                // if the applicant address is already taken by a member's delegateKey,
reset it to their member address
                if (members[memberAddressByDelegateKey[proposal.applicant]].exists) {
                    address memberToOverride =
memberAddressByDelegateKey[proposal.applicant];
                    memberAddressByDelegateKey[memberToOverride] = memberToOverride;
                    members[memberToOverride].delegateKey = memberToOverride;
                }

                // use applicant address as delegateKey by default
                members[proposal.applicant] = Member(proposal.applicant,
proposal.sharesRequested, proposal.lootRequested, true, 0, 0);
                memberAddressByDelegateKey[proposal.applicant] = proposal.applicant;
            }

            // mint new shares & loot
            totalShares = totalShares.add(proposal.sharesRequested);
            totalLoot = totalLoot.add(proposal.lootRequested);

            // if the proposal tribute is the first tokens of its kind to make it into
the guild bank, increment total guild bank tokens
            if (userTokenBalances[GUILD][proposal.tributeToken] == 0 &&
proposal.tributeOffered > 0) {
                totalGuildBankTokens += 1;
            }

            unsafeInternalTransfer(ESCROW, GUILD, proposal.tributeToken,
proposal.tributeOffered);
            unsafeInternalTransfer(GUILD, proposal.applicant, proposal.paymentToken,
proposal.paymentRequested);

            // if the proposal spends 100% of guild bank balance for a token, decrement
total guild bank tokens
            if (userTokenBalances[GUILD][proposal.paymentToken] == 0 &&
proposal.paymentRequested > 0) {
```

```
                totalGuildBankTokens -= 1;
            }

        // PROPOSAL FAILED
        } else {
            // return all tokens to the proposer (not the applicant, because funds come
from proposer)
            unsafeInternalTransfer(ESCROW, proposal.proposer, proposal.tributeToken,
proposal.tributeOffered);
        }

        _returnDeposit(proposal.sponsor);

        emit ProcessProposal(proposalIndex, proposalId, didPass);
    }

        //Helper functions

        function unsafeAddToBalance(address user, address token, uint256 amount)
internal {
            userTokenBalances[user][token] += amount;
            userTokenBalances[TOTAL][token] += amount;
        }

        function unsafeSubtractFromBalance(address user, address token, uint256 amount)
internal {
            userTokenBalances[user][token] -= amount;
            userTokenBalances[TOTAL][token] -= amount;
        }

        function unsafeInternalTransfer(address from, address to, address token,
uint256 amount) internal {
            unsafeSubtractFromBalance(from, token, amount);
            unsafeAddToBalance(to, token, amount);
        }

        function fairShare(uint256 balance, uint256 shares, uint256 totalShares)
internal pure returns (uint256) {
            require(totalShares != 0);

            if (balance == 0) { return 0; }
```

```
            uint256 prod = balance * shares;

            if (prod / balance == shares) { // no overflow in multiplication above?
                return prod / totalShares;
            }

            return (balance / totalShares) * shares;
        }
}



}
```