```
// ============================================================
// 1. MAUAX FOUNDERS NFT - GOVERNANCE TOKEN (ERC721)
// ============================================================

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

/**
 * @title MAUAX Founders NFT
 * @author GOS3 Team
 * @notice NFT de Governança para tokens físicos produzidos
 * @dev 100 NFTs: #001 Prefeito + #006-#031 Secretarias + #002-#005,#032-#100 Parceiros
 */
contract MauaxFoundersNFT is ERC721, ERC721URIStorage, Ownable, ReentrancyGuard {
    uint256 public constant MAX_SUPPLY = 100;
    uint256 private _currentTokenId = 0;

    mapping(uint256 => bool) public tokenExists;
    mapping(uint256 => string) public institutionalData;

    event TokenMinted(uint256 indexed tokenId, address indexed to, string institution);

    constructor() ERC721("MAUAX Founders Edition", "MAUAX-G") Ownable(msg.sender) {}

    function mintAllToTreasury() external onlyOwner {
        require(_currentTokenId == 0, "Already minted");

        // Token #001 - Pedra Fundamental
        _safeMint(owner(), 1);
        institutionalData[1] = "Prefeitura de Maua - Pedra Fundamental";

        // Tokens #006-#031 - Secretarias
        string[26] memory secretarias = [
            "Gabinete do Vice-Prefeito", "Camara Municipal de Maua",
            "Secretaria de Relacoes Institucionais", "Secretaria de Projetos Estrategicos",
            "Secretaria da Fazenda", "Secretaria de Meio Ambiente",
            "Secretaria de Desenvolvimento Economico", "Secretaria de Energia Social",
            "Secretaria de Obras e Planejamento", "Secretaria de Educacao",
            "Secretaria de Saude", "Secretaria de Mobilidade Urbana",
            "Secretaria de Assuntos Juridicos", "Secretaria de Administracao",
            "Secretaria de Comunicacao", "Secretaria de Cultura",
            "Secretaria de Esportes", "Secretaria de Habitacao",
            "Secretaria de Seguranca Publica", "Secretaria de Servicos Urbanos",
            "Secretaria de Assistencia Social", "Secretaria de Justica",
            "Secretaria de Inovacao e Tecnologia", "Secretaria de Financas",
            "Secretaria de Governo", "Secretaria de Cidadania e Acao Social"
        ];

        for (uint256 i = 0; i < 26; i++) {
            uint256 tokenId = i + 6;
            _safeMint(owner(), tokenId);
            institutionalData[tokenId] = secretarias[i];
        }

        // Demais tokens para parceiros
        for (uint256 i = 2; i <= 5; i++) {
            _safeMint(owner(), i);
            institutionalData[i] = "Parceiro Estrategico";
        }

        for (uint256 i = 32; i <= 100; i++) {
            _safeMint(owner(), i);
            institutionalData[i] = "Investidor DAO";
        }

        _currentTokenId = 100;
    }

    function tokenURI(uint256 tokenId) public view override(ERC721, ERC721URIStorage) returns
(string memory) {
        return super.tokenURI(tokenId);
    }

    function supportsInterface(bytes4 interfaceId) public view override(ERC721,
ERC721URIStorage) returns (bool) {
        return super.supportsInterface(interfaceId);
    }

    function _burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
        super._burn(tokenId);
    }

    function transferContractOwnership(address newOwner) external onlyOwner {
        transferOwnership(newOwner);
    }
```

```
}

// ==========================================================
// 2. MAUAX UTILITY TOKEN - MAIN CURRENCY (ERC20)
// ==========================================================

contract MauaxUtilityToken is ERC20, ERC20Burnable, Ownable {
    uint256 public constant MAX_SUPPLY = 100_000_000_000 * 10**18; // 100 bilhões
    uint256 public currentPhase = 0;

    mapping(address => bool) public authorizedMinters;

    event PhaseAdvanced(uint256 newPhase);
    event MinterAuthorized(address minter);
    event MinterRevoked(address minter);

    constructor() ERC20("MAUAX Utility Token", "MAUAX-C") Ownable(msg.sender) {}

    function mint(address to, uint256 amount) external onlyOwner {
        require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
        _mint(to, amount);
    }

    function authorizedMint(address to, uint256 amount) external {
        require(authorizedMinters[msg.sender], "Not authorized minter");
        require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
        _mint(to, amount);
    }

    function authorizeMinter(address minter) external onlyOwner {
        authorizedMinters[minter] = true;
        emit MinterAuthorized(minter);
    }

    function revokeMinter(address minter) external onlyOwner {
        authorizedMinters[minter] = false;
        emit MinterRevoked(minter);
    }

    function advancePhase() external onlyOwner {
        currentPhase++;
        emit PhaseAdvanced(currentPhase);
    }

    function transferContractOwnership(address newOwner) external onlyOwner {
        transferOwnership(newOwner);
    }
}

// ============================================================
// 3. MAUAX ENERGY TOKEN - RWA TOKEN (ERC20)
// ============================================================

contract MauaxEnergyToken is ERC20, ERC20Burnable, AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant ORACLE_ROLE = keccak256("ORACLE_ROLE");

    uint256 public constant ENERGY_PER_TOKEN = 1; // 1 token = 1 MWh
    uint256 public totalEnergyGenerated;

    mapping(address => uint256) public energyContributions;
    mapping(uint256 => EnergyBatch) public energyBatches;
    uint256 public batchCounter;

    struct EnergyBatch {
        uint256 amount;
        uint256 timestamp;
        string source;
        address validator;
    }

    event EnergyValidated(uint256 indexed batchId, uint256 amount, string source);
    event TokensMinted(address indexed to, uint256 amount, uint256 batchId);

    constructor() ERC20("MAUAX Energy Token", "MAUAX-E") {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(MINTER_ROLE, msg.sender);
        _grantRole(ORACLE_ROLE, msg.sender);
    }

    function validateAndMint(
        address to,
        uint256 energyAmount,
        string memory source
    ) external onlyRole(ORACLE_ROLE) {
        require(energyAmount > 0, "Invalid energy amount");

        batchCounter++;
        energyBatches[batchCounter] = EnergyBatch({
            amount: energyAmount,
            timestamp: block.timestamp,
            source: source,
            validator: msg.sender
        });

        uint256 tokensToMint = energyAmount * 10**decimals();
        _mint(to, tokensToMint);

        totalEnergyGenerated += energyAmount;
        energyContributions[to] += energyAmount;

        emit EnergyValidated(batchCounter, energyAmount, source);
        emit TokensMinted(to, tokensToMint, batchCounter);
    }
}

// ==========================================================
// 4. MAUAX RECYCLING TOKEN - CIRCULAR ECONOMY (ERC20)
// ==========================================================

contract MauaxRecyclingToken is ERC20, ERC20Burnable, AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant VALIDATOR_ROLE = keccak256("VALIDATOR_ROLE");

    uint256 public constant KG_PER_TOKEN = 1; // 1 token = 1kg reciclado
    uint256 public totalRecycledWeight;

    mapping(address => uint256) public recyclingContributions;
    mapping(address => bool) public authorizedCooperatives;

    struct RecyclingBatch {
        uint256 weight;
        uint256 timestamp;
        string materialType;
        address cooperative;
        address validator;
    }

    mapping(uint256 => RecyclingBatch) public recyclingBatches;
    uint256 public batchCounter;

    event MaterialValidated(uint256 indexed batchId, uint256 weight, string materialType);
    event CooperativeAuthorized(address cooperative);
    event CashbackPaid(address indexed recipient, uint256 amount);
    constructor() ERC20("MAUAX Recycling Token", "MAUAX-R") {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(MINTER_ROLE, msg.sender);
        _grantRole(VALIDATOR_ROLE, msg.sender);
    }

    function validateRecycling(
        address collector,
        uint256 weight,
        string memory materialType,
        address cooperative
    ) external onlyRole(VALIDATOR_ROLE) {
        require(authorizedCooperatives[cooperative], "Cooperative not authorized");
        require(weight > 0, "Invalid weight");

        batchCounter++;
        recyclingBatches[batchCounter] = RecyclingBatch({
            weight: weight,
            timestamp: block.timestamp,
            materialType: materialType,
            cooperative: cooperative,
            validator: msg.sender
        });

        uint256 tokensToMint = weight * 10**decimals();
        _mint(collector, tokensToMint);

        totalRecycledWeight += weight;
        recyclingContributions[collector] += weight;

        emit MaterialValidated(batchCounter, weight, materialType);
    }

    function authorizeCooperative(address cooperative) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        authorizedCooperatives[cooperative] = true;
        emit CooperativeAuthorized(cooperative);
    }
}

// ==========================================================
// 5. MAUAX SECURITY TOKEN TEMPLATE - INVESTMENT TOKENS
// ==========================================================

contract MauaxSecurityToken is ERC20, AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant WHITELISTED_INVESTOR =
keccak256("WHITELISTED_INVESTOR");

    uint256 public immutable MAX_SUPPLY;
    uint256 public immutable CAPEX_VALUE;
    uint256 public immutable EXPECTED_TIR;

    string public projectDescription;
    bool public isActive;

    mapping(address => uint256) public vestingSchedule;
    mapping(address => uint256) public dividendsOwed;

    event InvestorWhitelisted(address indexed investor);
    event DividendsDistributed(uint256 totalAmount);
    event TokensVested(address indexed investor, uint256 amount);

    constructor(
        string memory name,
        string memory symbol,
        uint256 maxSupply,
        uint256 capexValue,
        uint256 expectedTir,
        string memory description
    ) ERC20(name, symbol) {
        MAX_SUPPLY = maxSupply * 10**decimals();
        CAPEX_VALUE = capexValue;
        EXPECTED_TIR = expectedTir;
        projectDescription = description;
        isActive = true;

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(ADMIN_ROLE, msg.sender);
        _grantRole(MINTER_ROLE, msg.sender);
    }

    function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE) {
        require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
        require(hasRole(WHITELISTED_INVESTOR, to), "Investor not whitelisted");
        _mint(to, amount);
    }

    function whitelistInvestor(address investor) external onlyRole(ADMIN_ROLE) {
        grantRole(WHITELISTED_INVESTOR, investor);
        emit InvestorWhitelisted(investor);
    }

    function distributeDividends() external payable onlyRole(ADMIN_ROLE) {
        require(msg.value > 0, "No dividends to distribute");
        require(totalSupply() > 0, "No tokens in circulation");

        uint256 dividendPerToken = msg.value / totalSupply();

        // Implementation would iterate through holders
        emit DividendsDistributed(msg.value);
    }

    function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
        if (from != address(0) && to != address(0)) {
            require(hasRole(WHITELISTED_INVESTOR, to), "Recipient not whitelisted");
        }
        super._beforeTokenTransfer(from, to, amount);
    }
}

// ========================================================
// 6. MAUAX SEED NFT - PHASE 0 INVESTMENT (ERC721)
// ========================================================

contract MauaxSeedNFT is ERC721, Ownable {
    uint256 public constant TOKEN_ID = 1;
    uint256 public constant INVESTMENT_VALUE = 15_000_000; // R$ 15 milhões

    bool public minted = false;
    string private _tokenURI;

    event SeedNFTMinted(address indexed to, uint256 value);

    constructor() ERC721("MAUAX Seed Round Investment", "MAUAX-SEED")
Ownable(msg.sender) {
        _tokenURI = "https://app.mauax.com/metadata/seed/1.json";
    }

    function mintToSaleContract(address saleContractAddress) external onlyOwner {
        require(!minted, "SEED NFT already minted");
        require(saleContractAddress != address(0), "Invalid address");

        _safeMint(saleContractAddress, TOKEN_ID);
        minted = true;

        emit SeedNFTMinted(saleContractAddress, INVESTMENT_VALUE);
    }

    function tokenURI(uint256 tokenId) public view override returns (string memory) {
        require(tokenId == TOKEN_ID, "Token does not exist");
        return _tokenURI;
    }

    function updateTokenURI(string memory newURI) external onlyOwner {
        _tokenURI = newURI;
    }

    function transferContractOwnership(address newOwner) external onlyOwner {
        transferOwnership(newOwner);
    }
}

// ========================================================
// 7. INVESTOR VAULT - SEED GOVERNANCE (ACCESS CONTROL)
// ========================================================

contract InvestorVault is AccessControl, ReentrancyGuard {
    bytes32 public constant MEMBER_ROLE = keccak256("MEMBER_ROLE");

    IERC721 public immutable mauaxSeedNFT;
    uint256 public totalInvestment;
    uint256 public proposalCount;

    uint256 public constant VOTING_PERIOD = 3 days;
    uint256 public constant QUORUM_PERCENT = 51;

    mapping(address => uint256) public memberContributions;
    mapping(uint256 => Proposal) public proposals;

    struct Proposal {
        uint256 id;
        address target;
        bytes data;
        string description;
        uint256 executionTimestamp;
        bool executed;
        uint256 votesFor;
        mapping(address => bool) hasVoted;
    }

    event MemberAdded(address indexed member, uint256 contribution);
    event ProposalCreated(uint256 indexed id, address indexed proposer);
    event Voted(uint256 indexed proposalId, address indexed voter, uint256 weight);
    event ProposalExecuted(uint256 indexed proposalId);

    constructor(address _nftAddress) {
        mauaxSeedNFT = IERC721(_nftAddress);
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    function addMember(address member, uint256 contribution) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(MEMBER_ROLE, member);
        memberContributions[member] += contribution;
        totalInvestment += contribution;
        emit MemberAdded(member, contribution);
    }

    function createProposal(
        address target,
        bytes memory data,
        string memory description
    ) external onlyRole(MEMBER_ROLE) {
        proposalCount++;

        Proposal storage proposal = proposals[proposalCount];
        proposal.id = proposalCount;
        proposal.target = target;
        proposal.data = data;
        proposal.description = description;
        proposal.executionTimestamp = block.timestamp + VOTING_PERIOD;
        proposal.executed = false;
        proposal.votesFor = 0;

        emit ProposalCreated(proposalCount, msg.sender);
    }

    function vote(uint256 proposalId) external onlyRole(MEMBER_ROLE) {
        Proposal storage proposal = proposals[proposalId];
        require(block.timestamp < proposal.executionTimestamp, "Voting ended");
        require(!proposal.hasVoted[msg.sender], "Already voted");

        proposal.hasVoted[msg.sender] = true;
        uint256 voteWeight = memberContributions[msg.sender];
        proposal.votesFor += voteWeight;

        emit Voted(proposalId, msg.sender, voteWeight);
    }

    function executeProposal(uint256 proposalId) external nonReentrant {
        Proposal storage proposal = proposals[proposalId];
        require(block.timestamp >= proposal.executionTimestamp, "Voting not ended");
        require(!proposal.executed, "Already executed");

        uint256 quorum = (totalInvestment * QUORUM_PERCENT) / 100;
        require(proposal.votesFor >= quorum, "Quorum not reached");

        proposal.executed = true;
        (bool success, ) = proposal.target.call(proposal.data);
        require(success, "Execution failed");

        emit ProposalExecuted(proposalId);
    }

    function onERC721Received(address, address, uint256, bytes memory) public virtual returns
(bytes4) {
        return this.onERC721Received.selector;
    }
}

// =========================================================
// 8. MAUAX DAO TREASURY - SOVEREIGN FUND MANAGEMENT
// =========================================================

contract MauaxDAOTreasury is AccessControl, ReentrancyGuard {
    bytes32 public constant TREASURER_ROLE = keccak256("TREASURER_ROLE");
    bytes32 public constant PROPOSER_ROLE = keccak256("PROPOSER_ROLE");

    uint256 public totalFunds;
    uint256 public proposalCount;

    // Allocation percentages
    uint256 public constant INFRASTRUCTURE_PERCENT = 40;
    uint256 public constant SOCIAL_DEVELOPMENT_PERCENT = 30;
    uint256 public constant INNOVATION_PERCENT = 20;
    uint256 public constant RESERVE_PERCENT = 10;

    mapping(uint256 => TreasuryProposal) public treasuryProposals;
    mapping(address => uint256) public allocatedFunds;

    struct TreasuryProposal {
        uint256 id;
        string title;
        string description;
        uint256 amount;
        address recipient;
        uint8 category; // 1=Infrastructure, 2=Social, 3=Innovation, 4=Reserve
        bool executed;
        uint256 votesFor;
        uint256 votesAgainst;
        uint256 deadline;
    }

    event FundsReceived(address indexed from, uint256 amount, string source);
    event ProposalCreated(uint256 indexed id, string title, uint256 amount);
    event FundsAllocated(address indexed recipient, uint256 amount, uint8 category);

    constructor() {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(TREASURER_ROLE, msg.sender);
        _grantRole(PROPOSER_ROLE, msg.sender);
    }

    receive() external payable {
        totalFunds += msg.value;
        emit FundsReceived(msg.sender, msg.value, "Direct deposit");
    }

    function receiveFunds(string memory source) external payable {
        totalFunds += msg.value;
        emit FundsReceived(msg.sender, msg.value, source);
    }

    function createFundingProposal(
        string memory title,
        string memory description,
        uint256 amount,
        address recipient,
        uint8 category
    ) external onlyRole(PROPOSER_ROLE) {
        require(category >= 1 && category <= 4, "Invalid category");
        require(amount <= address(this).balance, "Insufficient funds");
```

```
        proposalCount++;
        treasuryProposals[proposalCount] = TreasuryProposal({
            id: proposalCount,
            title: title,
            description: description,
            amount: amount,
            recipient: recipient,
            category: category,
            executed: false,
            votesFor: 0,
            votesAgainst: 0,
            deadline: block.timestamp + 7 days
        });

        emit ProposalCreated(proposalCount, title, amount);
    }

    function executeFunding(uint256 proposalId) external onlyRole(TREASURER_ROLE)
nonReentrant {
        TreasuryProposal storage proposal = treasuryProposals[proposalId];
        require(!proposal.executed, "Already executed");
        require(block.timestamp >= proposal.deadline, "Voting period active");
        require(proposal.votesFor > proposal.votesAgainst, "Proposal rejected");

        proposal.executed = true;
        allocatedFunds[proposal.recipient] += proposal.amount;

        (bool success, ) = proposal.recipient.call{value: proposal.amount}("");
        require(success, "Transfer failed");

        emit FundsAllocated(proposal.recipient, proposal.amount, proposal.category);
    }

    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}

// ==================================================
// 9. ORACLE ENERGY DATA - IOT INTEGRATION
// ==================================================

contract OracleEnergyData is AccessControl {
    bytes32 public constant ORACLE_OPERATOR_ROLE =
keccak256("ORACLE_OPERATOR_ROLE");
```

```
bytes32 public constant DATA_PROVIDER_ROLE = keccak256("DATA_PROVIDER_ROLE");

MauaxEnergyToken public immutable energyToken;

struct EnergyReading {
    uint256 timestamp;
    uint256 generatedMWh;
    string sourceType;
    string location;
    address provider;
    bool validated;
}

mapping(uint256 => EnergyReading) public energyReadings;
mapping(address => bool) public authorizedSensors;
uint256 public readingCounter;

event ReadingSubmitted(uint256 indexed readingId, uint256 energy, string source);
event ReadingValidated(uint256 indexed readingId, address validator);
event TokensMinted(address indexed recipient, uint256 amount);

constructor(address _energyTokenAddress) {
    energyToken = MauaxEnergyToken(_energyTokenAddress);
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(ORACLE_OPERATOR_ROLE, msg.sender);
}

function submitEnergyReading(
    uint256 generatedMWh,
    string memory sourceType,
    string memory location
) external onlyRole(DATA_PROVIDER_ROLE) {
    require(generatedMWh > 0, "Invalid energy amount");

    readingCounter++;
    energyReadings[readingCounter] = EnergyReading({
        timestamp: block.timestamp,
        generatedMWh: generatedMWh,
        sourceType: sourceType,
        location: location,
        provider: msg.sender,
        validated: false
    });

    emit ReadingSubmitted(readingCounter, generatedMWh, sourceType);
    }

    function validateAndMintTokens(
        uint256 readingId,
        address recipient
    ) external onlyRole(ORACLE_OPERATOR_ROLE) {
        EnergyReading storage reading = energyReadings[readingId];
        require(!reading.validated, "Already validated");
        require(reading.generatedMWh > 0, "Invalid reading");

        reading.validated = true;

        // Mint tokens through the energy token contract
        energyToken.validateAndMint(recipient, reading.generatedMWh, reading.sourceType);

        emit ReadingValidated(readingId, msg.sender);
        emit TokensMinted(recipient, reading.generatedMWh);
    }

    function authorizeSensor(address sensor) external onlyRole(DEFAULT_ADMIN_ROLE) {
        authorizedSensors[sensor] = true;
        grantRole(DATA_PROVIDER_ROLE, sensor);
    }
}

// ========================================================
// 10. STAKING SYSTEM - DYNAMIC APY
// ========================================================

contract MauaxStakingSystem is AccessControl, ReentrancyGuard {
    bytes32 public constant STAKING_MANAGER_ROLE =
keccak256("STAKING_MANAGER_ROLE");

    MauaxUtilityToken public immutable stakingToken;

    struct StakeInfo {
        uint256 amount;
        uint256 startTime;
        uint256 lockPeriod;
        uint256 rewardRate;
        bool active;
    }

    mapping(address => StakeInfo[]) public userStakes;
    mapping(uint256 => uint256) public lockPeriodRates; // period => APY basis points
    uint256 public totalStaked;
    uint256 public rewardPool;
    uint256 public constant BASIS_POINTS = 10000;

    event Staked(address indexed user, uint256 amount, uint256 lockPeriod);
    event Unstaked(address indexed user, uint256 amount, uint256 reward);
    event RewardsDistributed(uint256 totalAmount);

    constructor(address _stakingTokenAddress) {
        stakingToken = MauaxUtilityToken(_stakingTokenAddress);
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(STAKING_MANAGER_ROLE, msg.sender);

        // Initialize lock periods with APY rates (in basis points)
        lockPeriodRates[30 days] = 500;   // 5% APY
        lockPeriodRates[90 days] = 800;   // 8% APY
        lockPeriodRates[180 days] = 1200; // 12% APY
        lockPeriodRates[365 days] = 1800; // 18% APY
    }

    function stake(uint256 amount, uint256 lockPeriod) external nonReentrant {
        require(amount > 0, "Invalid amount");
        require(lockPeriodRates[lockPeriod] > 0, "Invalid lock period");

        stakingToken.transferFrom(msg.sender, address(this), amount);

        userStakes[msg.sender].push(StakeInfo({
            amount: amount,
            startTime: block.timestamp,
            lockPeriod: lockPeriod,
            rewardRate: lockPeriodRates[lockPeriod],
            active: true
        }));

        totalStaked += amount;
        emit Staked(msg.sender, amount, lockPeriod);
    }

    function unstake(uint256 stakeIndex) external nonReentrant {
        StakeInfo storage userStake = userStakes[msg.sender][stakeIndex];
        require(userStake.active, "Stake not active");
        require(block.timestamp >= userStake.startTime + userStake.lockPeriod, "Lock period not
ended");
```

```
        uint256 reward = calculateReward(msg.sender, stakeIndex);
        uint256 totalAmount = userStake.amount + reward;

        userStake.active = false;
        totalStaked -= userStake.amount;

        require(stakingToken.balanceOf(address(this)) >= totalAmount, "Insufficient contract
balance");
        stakingToken.transfer(msg.sender, totalAmount);

        emit Unstaked(msg.sender, userStake.amount, reward);
    }

    function calculateReward(address user, uint256 stakeIndex) public view returns (uint256) {
        StakeInfo storage userStake = userStakes[user][stakeIndex];
        if (!userStake.active) return 0;

        uint256 timeStaked = block.timestamp - userStake.startTime;
        if (timeStaked < userStake.lockPeriod) return 0;

        uint256 annualReward = (userStake.amount * userStake.rewardRate) / BASIS_POINTS;
        return (annualReward * timeStaked) / 365 days;
    }

    function updateLockPeriodRate(uint256 lockPeriod, uint256 newRate) external
onlyRole(STAKING_MANAGER_ROLE) {
        lockPeriodRates[lockPeriod] = newRate;
    }

    function addRewards(uint256 amount) external onlyRole(STAKING_MANAGER_ROLE) {
        stakingToken.transferFrom(msg.sender, address(this), amount);
        rewardPool += amount;
        emit RewardsDistributed(amount);
    }

    function getUserStakeCount(address user) external view returns (uint256) {
        return userStakes[user].length;
    }
}

// ============================================================
// 11. MAUAX SEED SALE CONTRACT - PHASE 0 FUNDRAISING
// ============================================================

contract MauaxSeedSale is AccessControl, ReentrancyGuard {
    bytes32 public constant SALE_MANAGER_ROLE = keccak256("SALE_MANAGER_ROLE");

    MauaxSeedNFT public immutable seedNFT;
    InvestorVault public immutable investorVault;

    uint256 public constant TARGET_AMOUNT = 15_000_000 * 10**18; // R$ 15 milhões
(assuming 18 decimals)
    uint256 public constant MIN_INVESTMENT = 100_000 * 10**18;   // R$ 100 mil mínimo

    uint256 public totalRaised;
    uint256 public investorCount;
    bool public saleActive;
    bool public targetReached;

    mapping(address => uint256) public investments;
    address[] public investors;

    event InvestmentReceived(address indexed investor, uint256 amount);
    event TargetReached(uint256 totalAmount, uint256 investorCount);
    event SaleFinalized(address indexed vaultAddress);

    modifier onlyActiveSale() {
        require(saleActive, "Sale not active");
        require(!targetReached, "Target already reached");
        _;
    }

    constructor(address _seedNFTAddress, address _investorVaultAddress) {
        seedNFT = MauaxSeedNFT(_seedNFTAddress);
        investorVault = InvestorVault(_investorVaultAddress);

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(SALE_MANAGER_ROLE, msg.sender);

        saleActive = true;
    }

    function invest() external payable onlyActiveSale nonReentrant {
        require(msg.value >= MIN_INVESTMENT, "Below minimum investment");
        require(totalRaised + msg.value <= TARGET_AMOUNT, "Exceeds target amount");

        if (investments[msg.sender] == 0) {
            investors.push(msg.sender);
            investorCount++;
        }
        investments[msg.sender] += msg.value;
        totalRaised += msg.value;

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

        if (totalRaised >= TARGET_AMOUNT) {
            targetReached = true;
            emit TargetReached(totalRaised, investorCount);
        }
    }

    function finalizeSale() external onlyRole(SALE_MANAGER_ROLE) {
        require(targetReached, "Target not reached");
        require(saleActive, "Sale already finalized");

        saleActive = false;

        // Transfer NFT to investor vault
        seedNFT.safeTransferFrom(address(this), address(investorVault), 1);

        // Add all investors to the vault
        for (uint256 i = 0; i < investors.length; i++) {
            address investor = investors[i];
            investorVault.addMember(investor, investments[investor]);
        }

        emit SaleFinalized(address(investorVault));
    }

    function emergencyWithdraw() external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(!targetReached, "Cannot withdraw after target reached");

        saleActive = false;

        // Refund all investors
        for (uint256 i = 0; i < investors.length; i++) {
            address investor = investors[i];
            uint256 amount = investments[investor];
            if (amount > 0) {
                investments[investor] = 0;
                (bool success, ) = investor.call{value: amount}("");
                require(success, "Refund failed");
            }
        }
    }

    function getInvestorList() external view returns (address[] memory) {
        return investors;
    }

    function onERC721Received(address, address, uint256, bytes memory) public virtual returns
(bytes4) {
        return this.onERC721Received.selector;
    }
}

// =========================================================
// 12. MAUAX PSP INTEGRATION CONTRACT - PAYMENT GATEWAY
// =========================================================

contract MauaxPSPIntegration is AccessControl, ReentrancyGuard {
    bytes32 public constant PSP_OPERATOR_ROLE = keccak256("PSP_OPERATOR_ROLE");
    bytes32 public constant MERCHANT_ROLE = keccak256("MERCHANT_ROLE");

    MauaxUtilityToken public immutable utilityToken;
    MauaxRecyclingToken public immutable recyclingToken;
    MauaxEnergyToken public immutable energyToken;

    struct Transaction {
        uint256 id;
        address from;
        address to;
        uint256 amount;
        address tokenAddress;
        string transactionType;
        uint256 timestamp;
        bool completed;
    }

    mapping(uint256 => Transaction) public transactions;
    mapping(address => bool) public authorizedMerchants;
    mapping(address => uint256) public merchantBalances;

    uint256 public transactionCounter;
    uint256 public constant TRANSACTION_FEE_BASIS_POINTS = 20; // 0.2%

    event TransactionProcessed(uint256 indexed txId, address indexed from, address indexed to,
uint256 amount);
    event MerchantAuthorized(address indexed merchant);
event FeesCollected(uint256 amount);

constructor(
    address _utilityTokenAddress,
    address _recyclingTokenAddress,
    address _energyTokenAddress
) {
    utilityToken = MauaxUtilityToken(_utilityTokenAddress);
    recyclingToken = MauaxRecyclingToken(_recyclingTokenAddress);
    energyToken = MauaxEnergyToken(_energyTokenAddress);

    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(PSP_OPERATOR_ROLE, msg.sender);
}

function processPayment(
    address tokenAddress,
    address to,
    uint256 amount,
    string memory transactionType
) external nonReentrant returns (uint256) {
    require(amount > 0, "Invalid amount");
    require(to != address(0), "Invalid recipient");

    IERC20 token = IERC20(tokenAddress);

    uint256 fee = (amount * TRANSACTION_FEE_BASIS_POINTS) / 10000;
    uint256 netAmount = amount - fee;

    // Transfer tokens from sender
    token.transferFrom(msg.sender, address(this), amount);

    // Transfer net amount to recipient
    token.transfer(to, netAmount);

    // Keep fee in contract

    transactionCounter++;
    transactions[transactionCounter] = Transaction({
        id: transactionCounter,
        from: msg.sender,
        to: to,
        amount: amount,
        tokenAddress: tokenAddress,
        transactionType: transactionType,
            timestamp: block.timestamp,
            completed: true
        });

        emit TransactionProcessed(transactionCounter, msg.sender, to, netAmount);
        return transactionCounter;
    }

    function authorizeMerchant(address merchant) external onlyRole(PSP_OPERATOR_ROLE) {
        authorizedMerchants[merchant] = true;
        grantRole(MERCHANT_ROLE, merchant);
        emit MerchantAuthorized(merchant);
    }

    function generatePaymentQR(
        uint256 amount,
        address tokenAddress,
        string memory description
    ) external view onlyRole(MERCHANT_ROLE) returns (string memory) {
        // In practice, this would generate a QR code data string
        // For now, returning a placeholder
        return string(abi.encodePacked(
            "mauax://pay?amount=",
            Strings.toString(amount),
            "&token=",
            Strings.toHexString(uint160(tokenAddress), 20),
            "&merchant=",
            Strings.toHexString(uint160(msg.sender), 20)
        ));
    }

    function withdrawFees(address tokenAddress) external onlyRole(DEFAULT_ADMIN_ROLE)
nonReentrant {
        IERC20 token = IERC20(tokenAddress);
        uint256 balance = token.balanceOf(address(this));
        require(balance > 0, "No fees to withdraw");

        token.transfer(msg.sender, balance);
        emit FeesCollected(balance);
    }
}

// ========================================================
// 13. CROSS-CHAIN BRIDGE CONTRACT - ETHEREUM <> POLYGON
// ========================================================
contract MauaxCrossChainBridge is AccessControl, ReentrancyGuard {
  bytes32 public constant BRIDGE_OPERATOR_ROLE =
keccak256("BRIDGE_OPERATOR_ROLE");
  bytes32 public constant VALIDATOR_ROLE = keccak256("VALIDATOR_ROLE");

  mapping(address => bool) public supportedTokens;
  mapping(bytes32 => bool) public processedTransactions;
  mapping(address => uint256) public lockedBalances;

  uint256 public constant MIN_VALIDATORS = 3;
  uint256 public validatorCount;

  struct BridgeTransaction {
    bytes32 txHash;
    address token;
    address from;
    address to;
    uint256 amount;
    uint256 sourceChain;
    uint256 targetChain;
    uint256 timestamp;
    bool processed;
    uint256 validatorConfirmations;
  }

  mapping(bytes32 => BridgeTransaction) public bridgeTransactions;
  mapping(bytes32 => mapping(address => bool)) public transactionValidations;

  event TokensLocked(address indexed token, address indexed user, uint256 amount, bytes32
indexed txHash);
  event TokensReleased(address indexed token, address indexed user, uint256 amount,
bytes32 indexed txHash);
  event TransactionValidated(bytes32 indexed txHash, address indexed validator);

  constructor() {
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(BRIDGE_OPERATOR_ROLE, msg.sender);
    _grantRole(VALIDATOR_ROLE, msg.sender);
    validatorCount = 1;
  }

  function lockTokensForBridge(
    address token,
    uint256 amount,
    address targetAddress,
    uint256 targetChain
) external nonReentrant returns (bytes32) {
    require(supportedTokens[token], "Token not supported");
    require(amount > 0, "Invalid amount");

    IERC20(token).transferFrom(msg.sender, address(this), amount);
    lockedBalances[token] += amount;

    bytes32 txHash = keccak256(abi.encodePacked(
        token,
        msg.sender,
        targetAddress,
        amount,
        targetChain,
        block.timestamp,
        block.number
    ));

    bridgeTransactions[txHash] = BridgeTransaction({
        txHash: txHash,
        token: token,
        from: msg.sender,
        to: targetAddress,
        amount: amount,
        sourceChain: block.chainid,
        targetChain: targetChain,
        timestamp: block.timestamp,
        processed: false,
        validatorConfirmations: 0
    });

    emit TokensLocked(token, msg.sender, amount, txHash);
    return txHash;
}

function validateBridgeTransaction(bytes32 txHash) external onlyRole(VALIDATOR_ROLE) {
    require(!transactionValidations[txHash][msg.sender], "Already validated");

    BridgeTransaction storage transaction = bridgeTransactions[txHash];
    require(transaction.txHash == txHash, "Transaction not found");
    require(!transaction.processed, "Already processed");

    transactionValidations[txHash][msg.sender] = true;
    transaction.validatorConfirmations++;
```

```
        emit TransactionValidated(txHash, msg.sender);

        if (transaction.validatorConfirmations >= MIN_VALIDATORS) {
            _releaseBridgedTokens(txHash);
        }
    }

    function _releaseBridgedTokens(bytes32 txHash) internal {
        BridgeTransaction storage transaction = bridgeTransactions[txHash];
        require(!transaction.processed, "Already processed");

        transaction.processed = true;
        lockedBalances[transaction.token] -= transaction.amount;

        IERC20(transaction.token).transfer(transaction.to, transaction.amount);

        emit TokensReleased(transaction.token, transaction.to, transaction.amount, txHash);
    }

    function addSupportedToken(address token) external onlyRole(BRIDGE_OPERATOR_ROLE)
{
        supportedTokens[token] = true;
    }

    function addValidator(address validator) external onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(VALIDATOR_ROLE, validator);
        validatorCount++;
    }
}

// ========================================================
// 14. UNISWAP V3 INTEGRATION - AMM/DEX INTERFACE
// ========================================================

import "@uniswap/v3-periphery/contracts/interfaces/ISwapRouter.sol";
import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol";

contract MauaxDEXIntegration is AccessControl, ReentrancyGuard {
    bytes32 public constant DEX_MANAGER_ROLE = keccak256("DEX_MANAGER_ROLE");

    ISwapRouter public immutable swapRouter;
    IUniswapV3Factory public immutable factory;

    MauaxUtilityToken public immutable mauaxToken;
    mapping(address => address) public tokenPools;
    mapping(address => uint256) public liquidityPositions;

    uint24 public constant DEFAULT_POOL_FEE = 3000; // 0.3%

    event LiquidityAdded(address indexed token, uint256 amount0, uint256 amount1);
    event SwapExecuted(address indexed tokenIn, address indexed tokenOut, uint256 amountIn,
uint256 amountOut);

    constructor(
        address _swapRouterAddress,
        address _factoryAddress,
        address _mauaxTokenAddress
    ) {
        swapRouter = ISwapRouter(_swapRouterAddress);
        factory = IUniswapV3Factory(_factoryAddress);
        mauaxToken = MauaxUtilityToken(_mauaxTokenAddress);

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(DEX_MANAGER_ROLE, msg.sender);
    }

    function swapTokens(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 amountOutMinimum
    ) external nonReentrant returns (uint256 amountOut) {
        IERC20(tokenIn).transferFrom(msg.sender, address(this), amountIn);
        IERC20(tokenIn).approve(address(swapRouter), amountIn);

        ISwapRouter.ExactInputSingleParams memory params =
ISwapRouter.ExactInputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: DEFAULT_POOL_FEE,
            recipient: msg.sender,
            deadline: block.timestamp + 300,
            amountIn: amountIn,
            amountOutMinimum: amountOutMinimum,
            sqrtPriceLimitX96: 0
        });

        amountOut = swapRouter.exactInputSingle(params);
        emit SwapExecuted(tokenIn, tokenOut, amountIn, amountOut);
    }

    function createPool(address token0, address token1) external
onlyRole(DEX_MANAGER_ROLE) returns (address pool) {
        pool = factory.createPool(token0, token1, DEFAULT_POOL_FEE);
        tokenPools[token0] = pool;
        tokenPools[token1] = pool;
        return pool;
    }

    function getPoolAddress(address token0, address token1) external view returns (address) {
        return factory.getPool(token0, token1, DEFAULT_POOL_FEE);
    }
}

// ==========================================================
// 15. INSURANCE PROTOCOL - INFRASTRUCTURE COVERAGE
// ==========================================================

contract MauaxInsuranceProtocol is AccessControl, ReentrancyGuard {
    bytes32 public constant INSURANCE_MANAGER_ROLE =
keccak256("INSURANCE_MANAGER_ROLE");
    bytes32 public constant CLAIMS_ASSESSOR_ROLE =
keccak256("CLAIMS_ASSESSOR_ROLE");

    struct InsurancePolicy {
        uint256 policyId;
        address holder;
        string assetType;
        uint256 coverageAmount;
        uint256 premium;
        uint256 startDate;
        uint256 endDate;
        bool active;
    }

    struct Claim {
        uint256 claimId;
        uint256 policyId;
        address claimant;
        uint256 claimAmount;
        string description;
        uint256 submissionDate;
        uint8 status; // 0=Pending, 1=Approved, 2=Rejected, 3=Paid
        uint256 assessorVotes;
    }

    mapping(uint256 => InsurancePolicy) public policies;
    mapping(uint256 => Claim) public claims;
    mapping(uint256 => mapping(address => bool)) public claimVotes;

    uint256 public policyCounter;
    uint256 public claimCounter;
    uint256 public insurancePool;
    uint256 public totalCoverage;

    uint256 public constant MIN_ASSESSOR_VOTES = 3;
    uint256 public constant COVERAGE_RATIO = 80; // 80% coverage

    event PolicyCreated(uint256 indexed policyId, address indexed holder, uint256 coverage);
    event ClaimSubmitted(uint256 indexed claimId, uint256 indexed policyId, uint256 amount);
    event ClaimProcessed(uint256 indexed claimId, uint8 status, uint256 payout);

    constructor() {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(INSURANCE_MANAGER_ROLE, msg.sender);
        _grantRole(CLAIMS_ASSESSOR_ROLE, msg.sender);
    }

    function createPolicy(
        address holder,
        string memory assetType,
        uint256 coverageAmount,
        uint256 duration
    ) external onlyRole(INSURANCE_MANAGER_ROLE) returns (uint256) {
        uint256 premium = calculatePremium(coverageAmount, duration);
        require(insurancePool >= coverageAmount * COVERAGE_RATIO / 100, "Insufficient pool
funds");

        policyCounter++;
        policies[policyCounter] = InsurancePolicy({
            policyId: policyCounter,
            holder: holder,
            assetType: assetType,
            coverageAmount: coverageAmount,
            premium: premium,
            startDate: block.timestamp,
            endDate: block.timestamp + duration,
            active: true
        });

        totalCoverage += coverageAmount;
        emit PolicyCreated(policyCounter, holder, coverageAmount);
        return policyCounter;
    }

    function submitClaim(
        uint256 policyId,
        uint256 claimAmount,
        string memory description
    ) external returns (uint256) {
        InsurancePolicy storage policy = policies[policyId];
        require(policy.holder == msg.sender, "Not policy holder");
        require(policy.active, "Policy not active");
        require(block.timestamp <= policy.endDate, "Policy expired");
        require(claimAmount <= policy.coverageAmount, "Claim exceeds coverage");

        claimCounter++;
        claims[claimCounter] = Claim({
            claimId: claimCounter,
            policyId: policyId,
            claimant: msg.sender,
            claimAmount: claimAmount,
            description: description,
            submissionDate: block.timestamp,
            status: 0,
            assessorVotes: 0
        });

        emit ClaimSubmitted(claimCounter, policyId, claimAmount);
        return claimCounter;
    }

    function assessClaim(uint256 claimId, bool approve) external
onlyRole(CLAIMS_ASSESSOR_ROLE) {
        require(!claimVotes[claimId][msg.sender], "Already voted");

        Claim storage claim = claims[claimId];
        require(claim.status == 0, "Claim already processed");

        claimVotes[claimId][msg.sender] = true;
        if (approve) {
            claim.assessorVotes++;
        }
        // Auto-process if minimum votes reached
        if (claim.assessorVotes >= MIN_ASSESSOR_VOTES) {
            _processClaim(claimId, true);
        }
    }

    function _processClaim(uint256 claimId, bool approved) internal {
        Claim storage claim = claims[claimId];

        if (approved && insurancePool >= claim.claimAmount) {
            claim.status = 1; // Approved
            insurancePool -= claim.claimAmount;

            (bool success, ) = claim.claimant.call{value: claim.claimAmount}("");
            if (success) {
                claim.status = 3; // Paid
            }
        } else {
            claim.status = 2; // Rejected
        }

        emit ClaimProcessed(claimId, claim.status, approved ? claim.claimAmount : 0);
    }

    function calculatePremium(uint256 coverageAmount, uint256 duration) public pure returns
(uint256) {
        // Simple premium calculation: 2% of coverage per year
        return (coverageAmount * 2 * duration) / (100 * 365 days);
    }

    function addToPool() external payable onlyRole(INSURANCE_MANAGER_ROLE) {
        insurancePool += msg.value;
    }

    receive() external payable {
        insurancePool += msg.value;
    }
}
```