7. Token Transfer Functionality: Implement a function in the ERC-20 contract that allows a user to transfer tokens using the Batch ID.

Ans:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import "@openzeppelin/contracts/access/Ownable.sol";

// Custom ERC-20 token contract with expiry date

contract ExpirableERC20Token is ERC20, Ownable {

mapping(address => mapping(uint256 => uint256)) private userBatchBalances;

mapping(uint256 => uint256) private tokenExpiryDates;

uint256 private nextBatchId = 1;

event TokenMintedWithExpiry(address indexed to, uint256 value, uint256 expiryDate);

event TokensTransferredByBatch(address indexed from, address indexed to, uint256 batchId, uint256 value);

constructor(

string memory name,

string memory symbol,

address initialHolder,

uint256 initialSupply

) ERC20(name, symbol) Ownable(initialHolder) {

\_mint(initialHolder, initialSupply);

}

function mintWithExpiry(address to, uint256 value, uint256 expiryDate) external onlyOwner {

require(expiryDate > block.timestamp, "Expiry date must be in the future");

\_mint(to, value);

uint256 tokenId = \_tokenId(to, value);

tokenExpiryDates[tokenId] = expiryDate;

emit TokenMintedWithExpiry(to, value, expiryDate);

}

function transferByBatch(address to, uint256 value, uint256 batchId) external {

require(batchId > 0, "Batch ID must be greater than zero");

require(userBatchBalances[msg.sender][batchId] >= value, "Insufficient balance for the specified batch");

// Update user batch balances

userBatchBalances[msg.sender][batchId] -= value;

userBatchBalances[to][batchId] += value;

// Emit event for tokens transferred by batch

emit TokensTransferredByBatch(msg.sender, to, batchId, value);

// Transfer the tokens

\_transferTokens(msg.sender, to, value);

}

function getBatchBalance(uint256 batchId) external view returns (uint256) {

return userBatchBalances[msg.sender][batchId];

}

function isTokenValid(address owner, uint256 value) public view returns (bool) {

uint256 tokenId = \_tokenId(owner, value);

return (tokenExpiryDates[tokenId] == 0 || tokenExpiryDates[tokenId] > block.timestamp);

}

function \_transferTokens(address sender, address recipient, uint256 amount) internal {

\_beforeTokenTransfer(sender, recipient, amount);

super.\_transfer(sender, recipient, amount);

\_afterTokenTransfer(sender, recipient, amount);

}

function \_beforeTokenTransfer(address from, address to, uint256 amount) internal {

// Add any pre-transfer logic if needed

}

function \_afterTokenTransfer(address from, address to, uint256 amount) internal {

// Update user batch balances after a transfer

userBatchBalances[from][\_nextBatchId()] += amount;

userBatchBalances[to][\_nextBatchId()] += amount;

}

function \_nextBatchId() internal returns (uint256) {

return nextBatchId++;

}

function \_tokenId(address owner, uint256 value) internal pure returns (uint256) {

return uint256(keccak256(abi.encodePacked(owner, value)));

}

}