Apexia Token

Contract address:
0x4114Fe782fAb89c41855a230d4582b82F6885237

The ApexiaToken contract is an ERC20 token implementation with minting and burning capabilities, The contract leverages OpenZeppelin's ERC20 standard for token compliance and AccessControl for role-based access management. Tokens can be minted to increase supply or burned to reduce it. The contract ensures security by restricting these actions to authorized roles, and its deployment requires specifying addresses for both the admin and the minter. ApexiaToken integrates seamlessly into the Apexia ecosystem, facilitating efficient token management.

The primary goals of the contract are to ensure full compliance with the ERC20 standard for seamless interoperability with decentralized applications (dApps), implement fine-grained role-based access control for secure management of minting and burning tokens, and leverage audited, community-trusted libraries to minimize vulnerabilities and enhance overall security and trust.

Used Libraries

OpenZeppelin is a highly regarded and trusted library within the Ethereum ecosystem, widely used in the blockchain space for its reliability and security. Its contracts are rigorously audited to adhere to best practices, ensuring robust functionality and minimizing risks. By leveraging pre-audited and standardized implementations, OpenZeppelin significantly reduces the likelihood of introducing bugs or vulnerabilities, making it a preferred choice for secure and efficient smart contract development.

ERC20 Standard

Inheriting from OpenZeppelin's ERC20 ensures seamless interoperability with wallets and dApps while eliminating the need for manual implementation of ERC20 functions, thereby reducing potential bugs.

The ERC20 standard provides essential functionality, including token supply management, transfers (transfer and transferFrom), balance inquiries (balanceOf), and an allowance system (approve and allowance).

In ApexiaToken, the contract inherits all core ERC20 functionalities and extends them with custom logic through the mint and burn functions.

AccessControl

AccessControl simplifies role management by eliminating the need for manual, error-prone permission handling. Roles are assigned during deployment to ensure only trusted parties can mint or burn tokens.

It enables role-based access control for contract functions, with roles such as DEFAULT_ADMIN_ROLE for managing other roles and administrative privileges, and MINTER_ROLE for minting and burning tokens. AccessControl provides a scalable framework for managing permissions across multiple addresses without requiring custom logic.

In ApexiaToken, the constructor grants DEFAULT_ADMIN_ROLE to the defaultAdmin address, while the minter address is assigned the MINTER_ROLE to handle minting and burning operations.

Contract Functions

Readable functions

Function	Description
DEFAULT_ADMIN_ROLE	This is the default role in AccessControl, typically assigned to the contract deployer. It has the authority to manage all roles.
MINTER_ROLE	A custom role in the contract that allows holders to mint new tokens. Only accounts with this role can call minting functions.
allowance(owner address, spender address)	Returns the remaining number of tokens that a spender is allowed to spend on behalf of the token owner.
balanceOf(account address)	Returns the token balance of a specific address.
decimals	Indicates the number of decimal places the token uses. For example, if decimals is 18, a balance of 1 is 1 * 10^-18.
getRoleAdmin(role bytes32)	Returns the admin role that controls a specific role (e.g., DEFAULT_ADMIN_ROLE is the admin for MINTER_ROLE).
hasRole(role bytes32, account address)	Checks if a specific address holds a certain role (e.g., checks if an address has MINTER_ROLE).
name	Returns the name of the token (e.g., "ApexiaToken").
supportsInterface(interfaceId bytes4)	Used to check if the contract implements a certain interface (e.g., IERC20 or IAccessControl).
symbol	Returns the symbol of the token (e.g., "APX").
totalSupply	Returns the total number of tokens in circulation.

Executable Functions

Function	Description
approve(spender address, value uint256)	Allows a spender to spend a specified number of tokens on behalf of the token owner. This is used in conjunction with transferFrom.
burn(from address, amount uint256)	Allows a user or an entity with the necessary permission to destroy a specified number of tokens, reducing the total supply.
grantRole(role bytes32, account address)	Assigns a specific role (e.g., MINTER_ROLE) to a given address. Only accounts with the admin role for that specific role can call this function.
mint(to address, amount uint256)	Creates new tokens and assigns them to a specified address. This increases the total supply and can only be called by accounts with the MINTER_ROLE.
renounceRole(role bytes32, callerConfirmation address)	Allows a user to voluntarily relinquish a specific role they hold. This is useful if a user no longer wants to hold permissions.
revokeRole(role bytes32, account address)	Removes a specific role from an address. Only accounts with the admin role for the specific role can perform this action.
transfer(to address, value uint256)	Transfers a specified number of tokens from the caller's address to a recipient address.
transferFrom(from address, to address, value uint256)	Moves tokens from one address to another using the allowance mechanism. The caller must be approved by the token owner to spend the tokens.

Codes

Function name	**Code**
DEFAULT_ADMIN_ROLE	Public variable
MINTER_ROLE	Public variable
allowance(owner address, spender address)	`function allowance(address owner, address spender) public view virtual returns (uint256) { return _allowances[owner][spender]; }`
balanceOf(account address)	`function balanceOf(address account) public view virtual returns (uint256) { return _balances[account]; }`
decimals	`function decimals() public view virtual returns (uint8) { return 18; }`
getRoleAdmin(role bytes32)	`function getRoleAdmin(bytes32 role) public view virtual returns (bytes32) { return _roles[role].adminRole; }`
hasRole(role bytes32, account address)	`function hasRole(bytes32 role, address account) public view virtual returns (bool) { return _roles[role].hasRole[account]; }`
name	`function name() public view virtual returns (string memory) { return _name; }`
supportsInterface(interfaceId bytes4)	`function supportsInterface(bytes4 interfaceId) public view virtual returns (bool) { return interfaceId == type(IERC165).interfaceId; }`
symbol	`function symbol() public view virtual returns (string memory) { return _symbol; }`
totalSupply	`function totalSupply() public view virtual returns (uint256) { return _totalSupply; }`
approve(spender address, value uint256)	`function approve(address spender, uint256 value) public virtual returns (bool) { address owner = _msgSender(); _approve(owner, spender, value); return true; }`
burn(from address, amount uint256)	`function burn(address from, uint256 amount) public onlyRole(MINTER_ROLE) { _burn(from, amount); }`

Function name	**Code**

grantRole(role bytes32, account address)	function grantRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {
	        _grantRole(role, account);
	    }

mint(to address, amount uint256)	function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE) {
	        _mint(to, amount);
	    }

renounceRole(role bytes32, callerConfirmation address)	function renounceRole(bytes32 role, address callerConfirmation) public virtual {
	        if (callerConfirmation != _msgSender()) {
	            revert AccessControlBadConfirmation();
	        }
	
	        _revokeRole(role, callerConfirmation);
	    }

revokeRole(role bytes32, account address)	function revokeRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {
	        _revokeRole(role, account);
	    }

transfer(to address, value uint256)	function transfer(address to, uint256 value) public virtual returns (bool) {
	        address owner = _msgSender();
	        _transfer(owner, to, value);
	        return true;
	    }

transferFrom(from address, to address, value uint256)	function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
	        address spender = _msgSender();
	        _spendAllowance(from, spender, value);
	        _transfer(from, to, value);
	        return true;
	    }