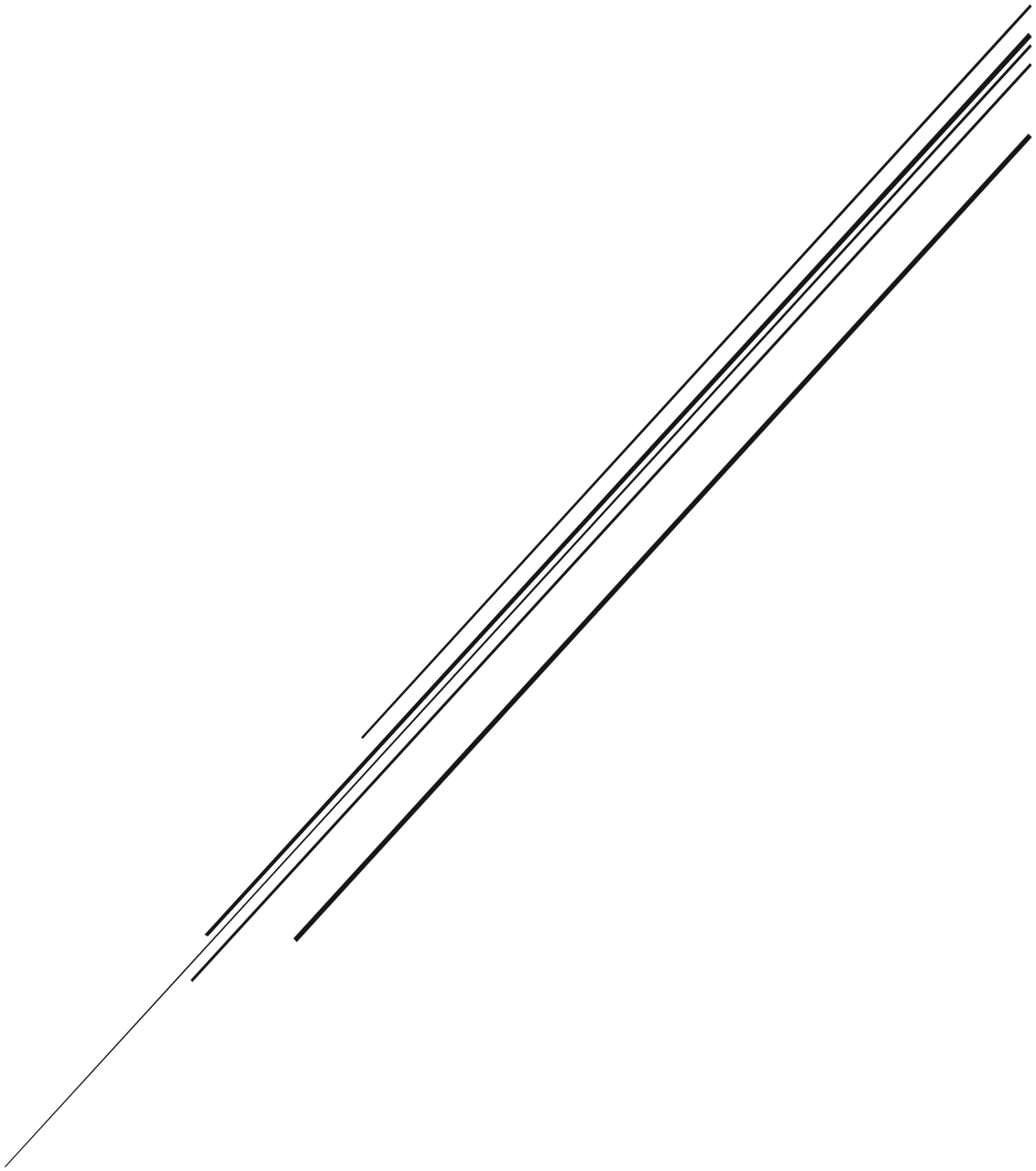
SMART CONTRACT & BLOCKCHAIN DOCUMENTATION

Detailed specifications of the smart contract - HYAX Token



HYDRAXIS

Nov 2024

Contents

Use case # 1: Initialize	
Title	Initialize Contract - Initialize( )
Description	The initialize function sets up the contract's state by initializing inherited contracts, minting tokens, and configuring essential parameters like token prices, limits, and addresses for oracles and tokens.
Actors and interfaces	• Deployer: The address that deploys and initializes the contract. • Inherited Contracts: Interfaces for ERC20, Ownable, Pausable, and ReentrancyGuard contracts.
Initial status and preconditions	• The initialize function can only be executed once due to the initializer modifier. • The deployer must have sufficient MATIC for transaction gas fees.
Basic Flow	
Step 1: Initialize inherited contracts using their respective initializer methods	
Step 2: Mint 500,000,000 HYAX tokens to the deployer's address.	
Step 3: Configure the HYAX token price to 0.006 USD with 8 decimals.	
Step 4: Set the minimum investment to $1 USD and maximum investment to $10,000 USD.	
Step 5: Set addresses for Whitelister and Treasury	
Step 6: Initialize price feed oracles for MATIC, USDC, USDT, WBTC, and WETH by setting: • Their respective token and price feed addresses. • Oracle interface instances (AggregatorV3Interface).	
Step 7: Implement ERC20 interfaces for each token (USDC, USDT, WBTC, WETH).	
Post Condition	
• The contract inherits the initialized states of ERC20, Ownable, ReentrancyGuard, and Pausable. • The deployer holds 500,000,000 HYAX tokens. • All key parameters (token prices, limits, addresses) are set and ready for interaction.	
Alternative flows	
• Transaction Reversion: If the function is called after the initial execution, the initializer modifier prevents re-execution. • Incorrect Address Setup: If invalid addresses are provided for tokens or oracles, the contract could fail to interact with those components.	

Executed by the smart contract implementer (Blockchain Director)

Development Environment -> Polygon Blockchain = Deployed Contract

Use case #2: Calculate HYAX Tokens for Investment	
Title	Calculate HYAX Tokens for Investment - calculateTotalHyaxTokenToReturn(uint256 _amount, uint256 _currentCryptocurrencyPrice)
Description	Determines the total USD value of an investment and calculates the corresponding amount of HYAX tokens based on the current cryptocurrency price.
Actors and interfaces	• Investor: The entity interacting with the function to estimate the token return. • Smart Contract: Provides the HYAX token price and contract balance.
Initial status and preconditions	• The investor provides valid _amount and _currentCryptocurrencyPrice. • The contract must have a sufficient HYAX token balance to fulfill the investment. • The calculated total investment in USD must meet or exceed the minimum investment threshold.
Basic Flow	
Step 1: Calculate the total investment in USD	
Step 2: Calculate the total HYAX tokens to return	
Step 3: Validate the investment meets the minimum threshold: • Ensure totalInvestmentInUsd is greater than or equal to minimumInvestmentAllowedInUSD. • If not, revert the transaction with an error message: "The amount to invest must be greater than the minimum established".	
Step 4: Validate token availability: • Ensure totalHyaxTokenToReturn is less than or equal to the HYAX token balance held by the contract. • If not, revert the transaction with an error message: "The investment made returns an amount of HYAX greater than the available".	
Step 5: Return the calculated totalInvestmentInUsd and totalHyaxTokenToReturn.	
Post Condition	
• The function only returns calculated values and does not alter the contract state. • The calculated values ensure compliance with investment thresholds and token availability.	
Alternative flows	
• Invalid Investment Amount: If totalInvestmentInUsd < minimumInvestmentAllowedInUSD, the transaction reverts with an error message. • Insufficient Contract Balance: If totalHyaxTokenToReturn > balanceOf(address(this)), the transaction reverts with an error message.	

Executed by the smart contract

Execution of the **investFromMatic()** or **investFromCryptoToken ( ) functionality** - > Function “Calculate total HYAX to return to investor” -> Return of value to the smart contract

Use case # 3: Add Investor to Whitelist	
Title	Add Investor to Whitelist - function addToWhiteList(address _investorAddress)
Description	Registers an investor's address to the whitelist, initializing their investment data.
Actors and interfaces	Smart Contract Owner or Whitelister: Authorized to add investors to the whitelist. Investor: The address being added to the whitelist.
Initial status and preconditions	• The caller must have the onlyOwnerOrWhitelister role. • The _investorAddress must be a valid, non-zero address. • The _investorAddress must not already be whitelisted.
Basic Flow	
Step 1: Validate _investorAddress: • Ensure _investorAddress is not the zero address (address(0)). • If invalid, revert with an error: "Investor address to add to the whitelist cannot be the zero address".	
Step 2: Check if the investor is already whitelisted: • Verify investorData[_investorAddress].isWhiteListed is false. • If already whitelisted, revert with an error: "That investor address has already been added to the whitelist".	
Step 3: Initialize a new InvestorData struct with default values: • Set isWhiteListed to true. • Set isBlacklisted to false. • Set isQualifiedInvestor to false. • Set totalHyaxBoughtByInvestor to 0. • Set totalUsdDepositedByInvestor to 0.	
Step 4: Store the initialized struct in the investorData mapping using _investorAddress as the key.	
Step 5: Emit the InvestorAddedToWhiteList event with msg.sender and _investorAddress as parameters.	
Post Condition	
• The _investorAddress is added to the whitelist with default values initialized in the investorData mapping. • The InvestorAddedToWhiteList event is logged.	

Alternative flows
• Invalid _investorAddress:If _investorAddress == address(0), the transaction reverts with: "Investor address to add to the whitelist cannot be the zero address". • Already Whitelisted Investor: If investorData[_investorAddress].isWhiteListed == true, the transaction reverts with: "That investor address has already been added to the whitelist".

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 4: Update Investor Whitelist Status	
Title	Update Investor Whitelist Status - updateWhitelistStatus(address _investorAddress,bool _newStatus)
Description	Updates the whitelist status of an investor to a new value.
Actors and interfaces	Smart Contract Owner or Whitelister: Authorized to update the whitelist status of an investor. Investor: The address whose whitelist status is being updated.
Initial status and preconditions	• The caller must have the onlyOwnerOrWhitelister role. • The _investorAddress must be a valid, non-zero address. • The _investorAddress must have a different current whitelist status than _newStatus.
Basic Flow	
Step 1: Validate _investorAddress: • Ensure _investorAddress is not the zero address (address(0)). • If invalid, revert with an error: "Investor address to update whitelist status cannot be the zero address".	
Step 2: Check if the current whitelist status matches _newStatus: • Verify that investorData[_investorAddress].isWhiteListed is not equal to _newStatus. • If already updated, revert with an error: "Investor address has already been updated to that status".	
Step 3: Update the whitelist status: • Set investorData[_investorAddress].isWhiteListed to _newStatus.	

Step 4: Emit the WhitelistStatusUpdated event with msg.sender, _investorAddress, and _newStatus as parameters.
Post Condition
• The whitelist status of _investorAddress is updated in the investorData mapping to _newStatus. • The WhitelistStatusUpdated event is logged.
Alternative flows
Invalid _investorAddress: • If _investorAddress == address(0), the transaction reverts with: "Investor address to update whitelist status cannot be the zero address". No Change in Whitelist Status: • If investorData[_investorAddress].isWhiteListed == _newStatus, the transaction reverts with: "Investor address has already been updated to that status".

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case #5: Update Investor Blacklist Status	
Title	Update Investor Blacklist Status - updateBlacklistStatus(address _investorAddress, bool _newStatus)
Description	Updates the blacklist status of an investor to a specified value.
Actors and interfaces	• Smart Contract Owner or Whitelister: Authorized to update the blacklist status. • Investor: The address whose blacklist status is being updated.
Initial status and preconditions	• The caller must have the onlyOwnerOrWhitelister role. • The _investorAddress must be a valid, non-zero address. • The _investorAddress must have a different current blacklist status than _newStatus.
Basic Flow	
Step 1: Validate _investorAddress: • Ensure _investorAddress is not the zero address (address(0)). • If invalid, revert with an error: "Investor address to update blacklist status cannot be the zero address".	

Step 2: Check if the current blacklist status matches _newStatus: • Verify that investorData[_investorAddress].isBlacklisted is not equal to _newStatus. • If already updated, revert with an error: "Investor address has already been updated to that status".
Step 3: Update the blacklist status: • Set investorData[_investorAddress].isBlacklisted to _newStatus.
Step 4: Emit the BlacklistStatusUpdated event with msg.sender, _investorAddress, and _newStatus as parameters.
Post Condition
• The blacklist status of _investorAddress is updated in the investorData mapping to _newStatus. • The BlacklistStatusUpdated event is logged.
Alternative flows
Invalid _investorAddress: • If _investorAddress == address(0), the transaction reverts with: "Investor address to update blacklist status cannot be the zero address". No Change in Blacklist Status: • If investorData[_investorAddress].isBlacklisted == _newStatus, the transaction reverts with: "Investor address has already been updated to that status".

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case #6: Update Qualified Investor Status	
Title	Update Qualified Investor Status - updateQualifiedInvestorStatus(address _qualifiedInvestorAddress, bool _newStatus)
Description	Updates the qualified investor status for a specified investor address.
Actors and interfaces	• Smart contract owner • Whitelister (authorized addresses that can update investor statuses)
Initial status and preconditions	• The caller must be the contract owner or an authorized whitelister. • The address _qualifiedInvestorAddress must: ○ Not be the zero address.

	○ Be already whitelisted (isWhiteListed must be true). ○ Not already have the same status as _newStatus. • The caller must have sufficient funds to pay for gas fees.
Basic Flow	
Step 1: • Check that _qualifiedInvestorAddress is not the zero address. If it is, revert the transaction with the error: "Investor address to update qualified investor status cannot be the zero address". • Check that _qualifiedInvestorAddress is already whitelisted (isWhiteListed == true). If not, revert the transaction with the error: "Investor address must be first added to the investor whitelist". • Check that the current status of _qualifiedInvestorAddress is not equal to _newStatus. If it is, revert the transaction with the error: "That investor address has already been updated to that status".	
Step 2: Update the isQualifiedInvestor property of the specified address to _newStatus.	
Step 3: Emit the QualifiedInvestorStatusUpdated event with the following details: • msg.sender: Address of the caller (owner or whitelister). • _qualifiedInvestorAddress: Address of the investor being updated. • _newStatus: The new qualified investor status.	
Post Condition	
• The isQualifiedInvestor status of the specified _qualifiedInvestorAddress is updated to _newStatus. • A QualifiedInvestorStatusUpdated event is emitted with the updated details.	
Alternative flows	
• Invalid address: If _qualifiedInvestorAddress is the zero address, the transaction reverts with an error message. • Address not whitelisted: If _qualifiedInvestorAddress is not whitelisted, the transaction reverts with an error message. • Redundant status update: If _newStatus matches the current status of the investor, the transaction reverts with an error message.	

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 7: Issue HYAX Tokens

Title	Issue HYAX Tokens - tokenIssuance(uint256 _amount)
Description	Issues a specified amount of HYAX tokens to the contract owner, ensuring the issuance adheres to defined limits.
Actors and interfaces	Smart contract owner.

Initial status and preconditions	• The caller must be the contract owner.
	• The _amount parameter must:
	  ○ Be at least 1 HYAX token.
	  ○ Not exceed 1,000,000,000 HYAX tokens in a single issuance.
	  ○ Not cause the total supply to exceed the maximum cap of 10,000,000,000 HYAX tokens.
	• The caller must have sufficient funds to pay for gas fees.

Basic Flow

Step 1: Input validation

1. Check that _amount is at least 1 HYAX token (_amount >= 10 ** decimals()). If not, revert the transaction with the error: "Amount of HYAX tokens to issue must be at least 1 token".
2. Check that _amount does not exceed 1,000,000,000 HYAX tokens (_amount <= 1000000000 * 10 ** decimals()). If it does, revert the transaction with the error: "Amount of HYAX tokens to issue at a time must be maximum 1000 M".
3. Check that the new total supply after issuance (totalSupply() + _amount) does not exceed 10,000,000,000 HYAX tokens. If it does, revert the transaction with the error: "Amount of HYAX tokens to issue surpasses the 10,000 M tokens".

Step 2: Mint tokens

- Call the _mint function to mint _amount HYAX tokens to the owner's address.

Step 3: Emit the TokenIssuance event with the following details:

- msg.sender: Address of the contract owner.
- _amount: The amount of HYAX tokens issued.

Post Condition

- The _amount of HYAX tokens is minted and added to the total supply.
- The minted tokens are transferred to the owner's address.
- A TokenIssuance event is emitted, indicating the successful issuance.

Alternative flows

Invalid _amount parameter:

- If _amount is less than 1 HYAX token, the transaction reverts with an error message.
- If _amount exceeds 1,000,000,000 HYAX tokens, the transaction reverts with an error message.

- If _amount causes the total supply to exceed 10,000,000,000 HYAX tokens, the transaction reverts with an error message.

Unauthorized caller:

- If the caller is not the contract owner, the transaction reverts due to the onlyOwner modifier.

Reentrancy attempt:

- If a reentrancy attack is attempted, the transaction is blocked due to the nonReentrant modifier.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 8: Validate and Track Investor's Investment	
Title	Validate and Track Investor's Investment - validateAndTrackInvestment(uint256 _totalInvestmentInUsd, address _investorAddress)
Description	Validates the investment amount in USD made by an investor and updates the total USD deposited by the investor, ensuring compliance with investment limits.
Actors and interfaces	Investors. Internal smart contract mechanism.
Initial status and preconditions	• _totalInvestmentInUsd must be a positive value. • _investorAddress must be a valid Ethereum address and should exist in the investorData mapping. • The maximumInvestmentAllowedInUSD variable must be properly set. • The caller must have sufficient gas to execute the transaction.
Basic Flow	
Step 1: Calculate the new total investment • Retrieve the investor's current total USD deposited (investorData[_investorAddress].totalUsdDepositedByInvestor). • Add _totalInvestmentInUsd to the current total to compute newTotalAmountInvestedInUSD**.**	
Step 2: Validate the investment limit. Check if newTotalAmountInvestedInUSD exceeds maximumInvestmentAllowedInUSD:	

• If it does, ensure the investor is a qualified investor (isQualifiedInvestor == true). • If not, revert the transaction with the error: "To buy that amount of HYAX its required to be a qualified investor".
Step 3: Update the investment record. Update investorData[_investorAddress].totalUsdDepositedByInvestor with newTotalAmountInvestedInUSD.
Post Condition
• The investor's total USD investment is updated in investorData. • The investment is validated to ensure compliance with the maximum allowed limit. • If applicable, only qualified investors are allowed to exceed the investment limit.
Alternative flows
Exceeding investment limit: • If newTotalAmountInvestedInUSD exceeds maximumInvestmentAllowedInUSD and the investor is not a qualified investor, the transaction reverts with an error message. Invalid investor address: • If _investorAddress does not exist in the investorData mapping, the behavior depends on additional logic (not specified in this snippet). Zero or negative investment amount: • If _totalInvestmentInUsd is zero or negative, the function will behave incorrectly unless additional input validation is added.

Executed by the smart contract

Execution from **investFromMatic( ), investFromCryptoToken functionality** - > Function " Validate and track investment " -> Return to the smart contract

Use case # 9: Get total supply	
Title	Get Total Supply - totalSupply()
Description	This function returns the total number of tokens currently in existence in the contract. It is a view function and does not alter the state of the contract.
Actors and interfaces	• External Users: Any external actor (such as an investor, user, or smart contract) that interacts with the token contract to query the total supply. • Smart Contract: The token contract itself provides this functionality for any actor requesting the total supply.

	• This function follows the ERC-20 standard for returning the total supply of tokens.
Initial status and preconditions	• The contract must be deployed and the total supply of tokens must be initialized when the contract is deployed or through a minting process. • The address calling this function does not need any specific permission or gas, as it is a read-only operation. However, the caller must be interacting with a valid instance of the token contract.
Basic Flow	
Step 1: The caller sends a query to the contract by calling totalSupply().	
Step 2: The contract retrieves the current value of the total supply of tokens from its internal state.	
Step 3: The contract returns the value of the total supply (as a uint256) to the caller.	
Step 4: The caller receives the total supply value and can use it for their purposes.	
Post Condition	
• The state of the contract remains unchanged as no state-modifying actions are performed. • The total supply of tokens remains the same. • The caller receives the total supply value as the output.	
Alternative flows	
Invalid Contract State: • If the contract is in an invalid state or has not been initialized with a total supply, the function may return a default value of zero or revert, depending on the contract implementation. Contract Does Not Implement totalSupply: • If the contract does not implement the totalSupply() function or is incompatible with the ERC-20 standard, a call to this function would fail with an error.	

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Executed by the Hydraxis platform to show information to investors

Website -> Connection to the smart contract using ethers.js -> Invocation of function execution using Alchemy API -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 10: Get the balance value of an address

Title	Balance Of - balanceOf(address account)
Description	This function returns the balance of tokens owned by a specific account (account). It is a view function and does not alter the state of the contract.
Actors and interfaces	• External Users: Any external actor (such as an investor, user, or smart contract) can query the balance of a specific account by calling this function. • Smart Contract: The token contract provides the balance of a given account. • ERC-20 Standard: This function follows the ERC-20 standard for querying the balance of tokens held by a specific account.
Initial status and preconditions	• The contract must be deployed and initialized with a valid total supply and accounts with balances. • The address calling this function does not need special permission, as it is a public read-only operation. • The account whose balance is being queried must exist within the system.
Basic Flow	
Step 1: The caller sends a query to the contract by calling balanceOf(account), passing the address of the account whose balance they wish to query.	
Step 2: The contract retrieves the balance of the specified account from its internal storage.	
Step 3: The contract returns the balance (in the form of a uint256) to the caller.	
Step 4: The caller receives the balance value of the specified account.	
Post Condition	
• The state of the contract remains unchanged as no state-modifying actions are performed. • The balance of the queried account remains the same. • The caller receives the balance of the account as the output**.**	
Alternative flows	
Invalid Account Address: • If the account address passed is invalid or does not exist, the contract will still return a balance of zero, as the balance of an uninitialized or non-existent address is treated as zero. • The function will not revert. Contract Does Not Implement balanceOf: • If the contract does not implement the balanceOf() function or is not ERC-20 compliant, the call will fail with an error.	

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Executed by the Hydraxis platform to show information to investors

Website -> Connection to the smart contract using ethers.js -> Invocation of function execution using Alchemy API -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 11: Transfer	
Title	Transfer Tokens - transfer(address to, uint256 value)
Description	This function transfers a specified value amount of tokens from the caller's account to a target account (to). It returns a boolean value indicating whether the transfer was successful and emits a Transfer event.
Actors and interfaces	• Sender (Caller): The account that is initiating the token transfer. This is the address calling the transfer() function. • Receiver (to): The account that will receive the transferred tokens. • Token Contract: The smart contract handling the token transfer and managing the token balances. • ERC-20 Standard: This function is part of the ERC-20 token standard, which includes functionality for transferring tokens between accounts.
Initial status and preconditions	• The caller must have a sufficient token balance to cover the transfer amount (value). • The contract must have been deployed with a valid token supply and initialized balances for accounts. • The address of the recipient (to) must be a valid Ethereum address. • The caller's account must not be frozen or restricted from performing transfers (if applicable). • The caller's account must be able to pay for the gas fees to execute the transaction.
Basic Flow	
Step 1: The caller initiates a transaction by calling the transfer(address to, uint256 value) function, specifying the recipient's address (to) and the number of tokens (value) to transfer.	

Step 2: The smart contract checks that the caller has a sufficient balance to transfer the specified amount.
Step 3: If the balance is sufficient, the contract deducts value tokens from the caller's balance and adds it to the to account's balance.
Step 4: A Transfer event is emitted to signal that the transfer has taken place, with the sender (msg.sender), receiver (to), and amount (value) as parameters.
Step 5: The function returns a true value, indicating that the transfer was successful.
Post Condition
The state of the contract is updated: • The caller's balance is reduced by the value. • The recipient's balance is increased by the value. A Transfer event is logged with the details of the transaction. The contract remains functional and ready for further interactions.
Alternative flows
Insufficient Balance: • If the caller's balance is insufficient to complete the transfer, the transaction will revert. The transfer will not occur, and the contract state remains unchanged. • The revert may also return an error such as "insufficient balance." Invalid Recipient Address: • If the recipient address (to) is invalid (e.g., the zero address 0x0), the transaction will revert. • A revert with an appropriate error message, such as "invalid address," may be triggered. Transfer Reverts Due to External Contract Restrictions: • If the recipient is a smart contract with a fallback function that reverts or does not accept tokens, the transfer will fail and revert. Gas Fees: • If the caller does not have enough gas to execute the transfer, the transaction will fail, and the transfer will not occur.

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 12: Get the value of the allowance

Title	Check Allowance for Token Spending - Allowance()

Description	This function returns the remaining number of tokens that a spender is allowed to spend on behalf of an owner. The allowance is set by the approve function and is updated when transferFrom is called.
Actors and interfaces	• Owner: The account that owns the tokens and has set the allowance for a spender. • Spender: The account authorized to spend tokens on behalf of the owner. • External Caller: Any user or contract querying the allowance. • ERC-20 Standard: This function is part of the ERC-20 token standard for querying the token allowance.
Initial status and preconditions	• The token contract must be deployed and properly initialized. • The owner address must have previously set an allowance for the spender using the approve function. • The owner and spender must be valid Ethereum addresses. • No specific gas is required, as this is a view function that does not modify the contract state.
Basic Flow	
Step 1: A user or contract calls the allowance(address owner, address spender) function, passing the owner address and spender address as parameters.	
Step 2: The smart contract retrieves the allowance value stored for the spender on behalf of the owner.	
Step 3: The function returns the allowance value (a uint256), which represents the maximum number of tokens the spender can still spend from the owner's balance.	
Post Condition	
• The state of the contract remains unchanged, as this is a view function. • The caller receives the current allowance value for the specified spender and owner combination.	
Alternative flows	
Owner or Spender Address Is Invalid: • If the owner or spender address is invalid (e.g., zero address 0x0), the function returns 0 because no allowance is set by default for an invalid address. Allowance Is Not Set: • If the owner has not set an allowance for the spender, the function returns 0 as allowances are initialized to zero by default. Allowance Updated: • If the approve or transferFrom function has been called prior to this query, the returned allowance reflects the updated value.	

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 13: Approve	
Title	Approve Allowance for Token Spending - approve(address spender, uint256 value)
Description	This function sets the number of tokens (value) a spender is allowed to spend on behalf of the caller. It returns a boolean indicating the success of the operation and emits an Approval event upon successful execution.
Actors and interfaces	• Token Holder: The account calling the function to authorize token spending. • Spender: The account authorized to spend tokens on behalf of the caller. • ERC-20 Standard: This function is part of the ERC-20 token standard for setting allowances.
Initial status and preconditions	• The caller must hold sufficient tokens if the spender attempts to transfer them later. • The spender address must be a valid Ethereum address. • The caller's account must have sufficient gas (in ETH/MATIC/etc.) to pay for the transaction execution. • If modifying an existing allowance, it is recommended to first set it to 0 to avoid race conditions.
Basic Flow	
Step 1: The token holder calls the approve function, passing the spender address and the desired allowance (value) as arguments.	
Step 2: The contract updates the allowance mapping to reflect the new value for the specified spender.	
Step 3: The contract emits an Approval event containing: • The token holder's address (msg.sender). • The spender's address. • The new allowance (value).	
Step 4: The function returns true to indicate the operation succeeded.	
Post Condition	
• The allowance for the specified spender is updated to the new value. • The Approval event is logged for external monitoring.	
Alternative flows	
Invalid spender Address:	

- If the spender is the zero address (0x0), the transaction is reverted. This prevents token loss or misallocation.

Race Condition Risk:

- If the allowance is updated without first setting it to 0, a race condition could allow a spender to spend both the old and new allowance due to transaction ordering. This is a known risk in the ERC-20 standard.

Allowance Already Set:

- If an allowance already exists, the function overwrites the existing value. The new allowance takes immediate effect.

Revert on Invalid Input:

- The transaction reverts if any other constraints (e.g., unexpected gas exhaustion) occur.

Event Emission for Tracking:

- In case of successful execution, third-party applications or services monitoring the blockchain can track the Approval event for real-time updates.

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 14: Transfer tokens From

Title	Transfer Tokens Using Allowance Mechanism - transferFrom(address from, address to, uint256 value)
Description	This function transfers a specified value of tokens from one account (from) to another (to) using the caller's allowance. It deducts the transferred amount from the caller's allowance and emits a Transfer event upon successful execution.
Actors and interfaces	• Spender: The account authorized to spend tokens on behalf of the from address (caller of the function). • Token Holder (from): The account from which tokens are transferred. • Recipient (to): The account receiving the transferred tokens. • ERC-20 Standard: This function adheres to the allowance and transfer mechanism defined in the ERC-20 token standard.

Initial status and preconditions	• The caller must have sufficient allowance granted by the from account to perform the transfer (value). • The from account must have a sufficient token balance to cover the transfer. • The to address must be a valid Ethereum address. • The caller's address must have sufficient gas (in ETH/MATIC/etc.) to execute the transaction.
Basic Flow	
Step 1: The caller invokes the transferFrom function, providing the from address, to address, and value of tokens to transfer.	
Step 2: The contract checks: • If the value is less than or equal to the allowance granted to the caller by the from address. • If the from account has a sufficient token balance for the transfer.	
Step 3: The contract deducts the value from the from account's token balance.	
Step 4: The contract adds the value to the to account's token balance.	
Step 5: The allowance for the caller is reduced by value.	
Step 6: A Transfer event is emitted, logging the from address, to address, and transferred value.	
Step 7: The function returns true to indicate successful execution.	
Post Condition	
• The token balance of the from account is reduced by the value. • The token balance of the to account is increased by the value. • The allowance for the caller is reduced by the value. • A Transfer event is emitted and recorded on the blockchain.	
Alternative flows	
Insufficient Allowance: • If the value exceeds the allowance, the transaction is reverted, and no changes occur. Insufficient Token Balance: • If the from account's token balance is insufficient, the transaction is reverted, and no changes occur. Invalid to Address: • If the to address is the zero address (0x0), the transaction is reverted to prevent token loss. Zero Value Transfer: • If the value is zero, the function completes successfully but only emits a Transfer event reflecting a transfer of zero tokens. Gas Limit Exceeded: • If the transaction exceeds the gas limit, the function execution is halted, and all state changes are reverted. No Allowance Set: • If no allowance exists for the caller by the from account, the transaction is reverted.	

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 15: Invest from Matic	
Title	Invest in HYAX Using MATIC - investFromMatic()
Description	Allows whitelisted investors to invest in HYAX tokens using MATIC. Transfers MATIC to the treasury and returns HYAX tokens to the investor.
Actors and interfaces	**I**nvestor: A whitelisted wallet interacting with the contract.

Initial status and preconditions	• The address executing the function:
	  ○ Is on the whitelist and not on the blacklist.
	  ○ Has sufficient MATIC balance to cover the investment and gas fees.
	• The contract has sufficient HYAX token supply to fulfill the investment.
	• The function must be executed in compliance with the nonReentrant modifier to prevent reentrancy attacks.

Basic Flow	
Step 1: MATIC sent with the transaction is automatically transferred to the contract using the payable modifier.	
Step 2: The function calculates the amount of HYAX tokens to return using the current HYAX price for MATIC.	
Step 3: Validates the total investment amount in USD. Updates the investor's investment data.	
Step 4: Sends the MATIC received to the treasury address.	
Step 5: Transfers the calculated amount of HYAX tokens to the investor's wallet.	
Step 6: Updates the total amount of HYAX tokens purchased by the investor.	
Step 7: Emits the InvestFromMatic event, logging the investor's address, MATIC amount, investment in USD, and HYAX tokens purchased.	
Step 8: Returns true, indicating a successful transaction.	
Post Condition	
• The MATIC amount is transferred to the treasury address. • The investor's wallet receives the calculated HYAX tokens. • The investor's total HYAX purchase record is updated. • The InvestFromMatic event is emitted with transaction details.	
Alternative flows	
Invalid Parameters:	

- If the investor is not on the whitelist or is blacklisted, the function reverts due to the investorWhitelistAndBlacklistCheck modifier.

Insufficient HYAX Tokens:

- If the contract does not have enough HYAX tokens to fulfill the investment, the transaction reverts.

MATIC Transfer Failure:

- If transferring MATIC to the treasury address fails, the transaction reverts with the error:
  "There was an error on sending the MATIC investment to the treasury".

HYAX Token Transfer Failure:

- If transferring HYAX tokens to the investor fails, the transaction reverts with the error:
  "There was an error on sending back the HYAX Token to the investor".

Executed by a user with a crypto wallet

Website -> Query of the USD value of the MATC to be displayed in the front -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Query of the MATIC USD value with oracle on Blockchain -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 15: Invest from Crypto Token	
Title	Invest in HYAX Using a Specified Cryptocurrency - investFromCryptoToken(TokenType tokenType,uint256 _amount)
Description	Allows whitelisted investors to invest in HYAX tokens using a supported cryptocurrency. Transfers the specified cryptocurrency to the treasury and returns HYAX tokens to the investor.
Actors and interfaces	A whitelisted wallet interacting with the contract.

Initial status and preconditions	• The address executing the function:
	  ○ Is on the whitelist and not on the blacklist.
	  ○ Holds the specified cryptocurrency (*e.g., USDC, USDT, WBTC, WETH*) in sufficient amount to cover the investment.
	• The contract has sufficient HYAX token supply to fulfill the investment.
	• The function must be executed in compliance with the nonReentrant modifier to prevent reentrancy attacks.

	• The approve function must have been called by the investor, authorizing the transfer of the specified cryptocurrency.
Basic Flow	
Step 1: Determine the cryptocurrency type and fetch the associated token contract and its current price by calling getCurrentTokenPrice. If the token type is invalid, revert with the error "Invalid token type".	
Step 2: Calculate the total HYAX tokens to return using the provided amount and the token price. Validate that the investment meets the minimum requirements and that there are sufficient HYAX tokens available for sale.	
Step 3: Update and validate the investor's data atomically using validateAndTrackInvestment.	
Step 4: Transfer the specified cryptocurrency from the investor to this contract using transferFrom. Revert the transaction if the transfer fails with the error: "There was an error on receiving the token investment".	
Step 5: Transfer the cryptocurrency from the contract to the treasury address. Revert the transaction if this step fails with the error: "There was an error on sending the token investment to the treasury".	
Step 6: Transfer the calculated HYAX tokens to the investor's wallet. Revert the transaction if this step fails with the error: "There was an error on sending back the HYAX Token to the investor".	
Step 7: Update the total amount of HYAX tokens purchased by the investor.	
Step 8: Emit the InvestFromCryptoToken event, logging the token type, investor address, investment amount, total investment in USD, and HYAX tokens purchased.	
Step 9: Return true, indicating the transaction's success.	
Post Condition	
• The specified cryptocurrency amount is transferred to the treasury address. • The investor's wallet receives the calculated HYAX tokens. • The investor's total HYAX purchase record is updated. • The InvestFromCryptoToken event is emitted with transaction details.	
Alternative flows	
Invalid Token Type: • If an unsupported token type is passed, the function reverts with the error: "Invalid token type". Insufficient Approval: • If the investor has not approved the contract to transfer the specified cryptocurrency, the transaction reverts due to the transferFrom function. Token Transfer Failure: • If the cryptocurrency transfer from the investor to the contract fails, the transaction reverts with the error: "There was an error on receiving the token investment". Treasury Transfer Failure:	

- If the cryptocurrency transfer from the contract to the treasury fails, the transaction reverts with the error:
  "There was an error on sending the token investment to the treasury".

HYAX Token Transfer Failure:

- If the transfer of HYAX tokens to the investor fails, the transaction reverts with the error:
  "There was an error on sending back the HYAX Token to the investor".

Executed by a user with a crypto wallet

Website -> Query of the Crypto Token value of the Crypto Token to be displayed in the front -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet - > Query of the USD value of the Crypto Token with oracle on Blockchain -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 16: Update HYAX Price	
Title	Update the Price of HYAX Tokens in USD - updateHyaxPrice(uint256 _newHyaxPrice)
Description	Allows the contract owner to update the price of HYAX tokens in USD, ensuring it falls within the specified range and avoids redundant updates.
Actors and interfaces	Smart Contract Owner: The only entity authorized to execute the function.

Initial status and preconditions	• The caller must be the contract owner, verified by the onlyOwner modifier.
	• The _newHyaxPrice must:
	  • Differ from the current hyaxPrice.
	  • Be within the range: **0.005 USD** (500,000 in 8 decimals) to **1,000 USD** (100,000,000,000 in 8 decimals).
	• The contract must emit the UpdatedHyaxPrice event upon a successful update.

Basic Flow	
Step 1: Verify that _newHyaxPrice is not equal to the current hyaxPrice. If they are the same, revert with the error: "HYAX price has already been modified to that value".	
Step 2: Validate that _newHyaxPrice is at least 0.005 USD (500,000 with 8 decimals). If not, revert with the error: "Price of HYAX token must be at least USD 0.005, that is 500000 with 8 decimals".	

Step 3: Validate that _newHyaxPrice does not exceed 1,000 USD (100,000,000,000 with 8 decimals). If it exceeds, revert with the error: "Price of HYAX token must be at maximum USD 1000, that is 100000000000 with 8 decimals".
Step 4: Update the hyaxPrice state variable with the value of _newHyaxPrice.
Step 5: Emit the UpdatedHyaxPrice event, logging the new HYAX price.
Post Condition
The hyaxPrice state variable is updated to the new value specified by _newHyaxPrice. The UpdatedHyaxPrice event is emitted, signaling the successful price update.
Alternative flows
Same Price Update: • If _newHyaxPrice equals the current hyaxPrice, the transaction reverts with the error: "HYAX price has already been modified to that value". Price Below Minimum: • If _newHyaxPrice is less than 0.005 USD (500,000 with 8 decimals), the transaction reverts with the error: "Price of HYAX token must be at least USD 0.005, that is 500000 with 8 decimals". Price Above Maximum: • If _newHyaxPrice exceeds 1,000 USD (100,000,000,000 with 8 decimals), the transaction reverts with the error: "Price of HYAX token must be at maximum USD 1000, that is 100000000000 with 8 decimals". Unauthorized Caller: • If the function is called by an address other than the contract owner, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 17: Update Minimum Investment Allowed in USD

Title	Update the Minimum Investment Amount in USD - updateMinimumInvestmentAllowedInUSD(uint256 _newMinimumInvestmentAllowedInUSD)

Description	Allows the contract owner to set a new minimum investment amount in USD, ensuring the value is greater than zero, not redundant, and does not exceed the maximum investment limit.
Actors and interfaces	Smart Contract Owner: The only entity authorized to execute this function.
Initial status and preconditions	The caller must be the contract owner, verified by the onlyOwner modifier. The _newMinimumInvestmentAllowedInUSD must: • Be greater than zero. • Differ from the current minimumInvestmentAllowedInUSD. • Be less than or equal to maximumInvestmentAllowedInUSD.
Basic Flow	
Step 1: Verify that _newMinimumInvestmentAllowedInUSD is greater than zero. If not, revert with the error: "New minimum amount to invest must be greater than zero".	
Step 2: Check that _newMinimumInvestmentAllowedInUSD differs from the current minimumInvestmentAllowedInUSD. If they are the same, revert with the error: "Minimum investment allowed in USD has already been modified to that value".	
Step 3: Ensure _newMinimumInvestmentAllowedInUSD is less than or equal to maximumInvestmentAllowedInUSD. If it exceeds, revert with the error: "New minimum amount to invest must be less than the maximum investment allowed".	
Step 4: Update the minimumInvestmentAllowedInUSD state variable to _newMinimumInvestmentAllowedInUSD.	
Step 5: Emit the UpdatedMinimumInvestmentAllowedInUSD event, logging the updated minimum investment amount.	
Post Condition	
• The minimumInvestmentAllowedInUSD variable is updated to the new value specified by _newMinimumInvestmentAllowedInUSD. • The UpdatedMinimumInvestmentAllowedInUSD event is emitted to signal the successful update.	
Alternative flows	
New Minimum Investment Is Zero or Negative: • If _newMinimumInvestmentAllowedInUSD is less than or equal to zero, the transaction reverts with the error: "New minimum amount to invest must be greater than zero". Same Value as Current Minimum Investment: • If _newMinimumInvestmentAllowedInUSD equals the current minimumInvestmentAllowedInUSD, the transaction reverts with the error: "Minimum investment allowed in USD has already been modified to that value".	

Exceeds Maximum Investment Allowed:

- If _newMinimumInvestmentAllowedInUSD is greater than maximumInvestmentAllowedInUSD, the transaction reverts with the error:
  "New minimum amount to invest must be less than the maximum investment allowed".

Unauthorized Caller:

- If the function is called by any address other than the contract owner, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 18: Update Maximum Investment Allowed in USD

Title	Update the Maximum Investment Amount in USD - updateMaximumInvestmentAllowedInUSD(uint256 _newMaximumInvestmentAllowedInUSD)
Description	Allows the contract owner to set a new maximum investment amount in USD for non-qualified investors. Ensures the value is greater than zero, not redundant, and at least equal to the minimum investment limit.
Actors and interfaces	Smart Contract Owner: The only entity authorized to execute this function.
Initial status and preconditions	The caller must be the contract owner, enforced by the onlyOwner modifier. The _newMaximumInvestmentAllowedInUSD parameter must: • Be greater than zero. • Differ from the current maximumInvestmentAllowedInUSD. • Be greater than or equal to minimumInvestmentAllowedInUSD.
Basic Flow	
Step 1: Verify that _newMaximumInvestmentAllowedInUSD is greater than zero. If not, revert with the error: "New maximum amount to invest, must be greater than zero".	

Step 2: Check that _newMaximumInvestmentAllowedInUSD is not equal to the current maximumInvestmentAllowedInUSD. If they are the same, revert with the error: "New maximum amount to invest, has already been modified to that value".
Step 3: Ensure _newMaximumInvestmentAllowedInUSD is greater than or equal to minimumInvestmentAllowedInUSD. If it is less, revert with the error: "New maximum amount to invest, must be greater than the minimum investment allowed".
Step 4: Update the maximumInvestmentAllowedInUSD state variable to _newMaximumInvestmentAllowedInUSD.
Step 5: Emit the UpdatedMaximumInvestmentAllowedInUSD event, logging the updated maximum investment amount.
Post Condition
The maximumInvestmentAllowedInUSD variable is updated to the new value specified by _newMaximumInvestmentAllowedInUSD. The UpdatedMaximumInvestmentAllowedInUSD event is emitted, signaling the successful update.
Alternative flows
New Maximum Investment is Zero or Negative: • If _newMaximumInvestmentAllowedInUSD is less than or equal to zero, the transaction reverts with the error: "New maximum amount to invest, must be greater than zero". Same Value as Current Maximum Investment: • If _newMaximumInvestmentAllowedInUSD equals the current maximumInvestmentAllowedInUSD, the transaction reverts with the error: "New maximum amount to invest, has already been modified to that value". Less Than Minimum Investment: • If _newMaximumInvestmentAllowedInUSD is less than minimumInvestmentAllowedInUSD, the transaction reverts with the error: "New maximum amount to invest, must be greater than the minimum investment allowed". Unauthorized Caller: • If the function is called by any address other than the contract owner, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case #14: Update Whitelister Address

Title	Update the Whitelister Address - updateWhiteListerAddress(address _newWhiteListerAddress)
Description	Allows the contract owner to update the address designated as the whitelister. The new address must not be the zero address and must be different from the current whitelister address.
Actors and interfaces	Smart Contract Owner: The only entity authorized to execute this function.
Initial status and preconditions	The caller must be the contract owner, enforced by the onlyOwner modifier. The _newWhiteListerAddress parameter must: • Be a valid non-zero Ethereum address. • Differ from the current whiteListerAddress.

Basic Flow

Step 1: Check that _newWhiteListerAddress is not the zero address (address(0)). If it is, revert with the error:
"The whitelister address cannot be the zero address".

Step 2: Verify that _newWhiteListerAddress is different from the current whiteListerAddress. If they are the same, revert with the error:
"whitelister address has already been modified to that value".

Step 3: Update the whiteListerAddress state variable to the value of _newWhiteListerAddress.

Step 4: Emit the UpdatedWhiteListerAddress event, logging the new whitelister address.

Post Condition

- The whiteListerAddress variable is updated to the new value specified by _newWhiteListerAddress.
- The UpdatedWhiteListerAddress event is emitted, signaling the successful update.

Alternative flows

Zero Address Provided:

- If _newWhiteListerAddress is the zero address (address(0)), the transaction reverts with the error:
  "The whitelister address cannot be the zero address".

Same Address as Current Whitelister:

- If _newWhiteListerAddress equals the current whiteListerAddress, the transaction reverts with the error:
  "whitelister address has already been modified to that value".

Unauthorized Caller:

- If the function is called by any address other than the contract owner, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 19: Update Treasury Address	
Title	Update the Treasury Address - updateTreasuryAddress(address _newTreasuryAddress)
Description	Allows the contract owner to update the treasury address. The new address must not be the zero address and must differ from the current treasury address.
Actors and interfaces	Smart Contract Owner: The only authorized user to execute this function.
Initial status and preconditions	The caller must be the contract owner, enforced by the onlyOwner modifier. The _newTreasuryAddress parameter must: • Be a valid non-zero Ethereum address. • Differ from the current treasuryAddress.
Basic Flow	
Step 1: Verify that _newTreasuryAddress is not the zero address (address(0)). If it is, revert with the error message: "The treasury address cannot be the zero address".	
Step 2: Ensure _newTreasuryAddress is not equal to the current treasuryAddress. If they are the same, revert with the error message: "Treasury address has already been modified to that value".	
Step 3: Update the treasuryAddress state variable to the value of _newTreasuryAddress.	
Step 4: Emit the UpdatedTreasuryAddress event, logging the updated treasury address.	
Post Condition	
The treasuryAddress variable is updated to the value of _newTreasuryAddress. The UpdatedTreasuryAddress event is emitted, signaling a successful update.	
Alternative flows	
Zero Address Provided: • If _newTreasuryAddress is the zero address (address(0)), the transaction reverts with the error: "The treasury address cannot be the zero address". Same Address as Current Treasury:	

- If _newTreasuryAddress equals the current treasuryAddress, the transaction reverts with the error:
  "Treasury address has already been modified to that value".

Unauthorized Caller:

- If a non-owner address attempts to execute the function, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 20: Update Token Address	
Title	Update the Address of a Specific Token - updateTokenAddress(TokenType tokenType,address newTokenAddress)
Description	This function allows the contract owner to update the address of a specified token (USDC, USDT, WBTC, or WETH).
Actors and interfaces	Smart Contract Owner: The only authorized user to execute this function.
Initial status and preconditions	The caller must be the contract owner, enforced by the onlyOwner modifier. The newTokenAddress parameter must: • Be a valid non-zero Ethereum address. • Differ from the current address of the specified token type. The tokenType must be one of the predefined types (USDC, USDT, WBTC, WETH).
Basic Flow	
Step 1: Validate that newTokenAddress is not the zero address (address(0)). If it is, revert with the error message: "The token address cannot be the zero address".	
Step 2: Identify the tokenType parameter. • If tokenType is USDC: 1. Verify that newTokenAddress is different from usdcTokenAddress. ▪ If not, revert with: "USDC token address has already been modified to that value". 2. Update usdcTokenAddress with newTokenAddress. 3. Assign usdcToken to the new token contract instance (IERC20(newTokenAddress)).	

4. Emit the UpdatedUsdcTokenAddress event.

- If tokenType is USDT:
  1. Verify that newTokenAddress is different from usdtTokenAddress.
     - If not, revert with: "USDT token address has already been modified to that value".
  2. Update usdtTokenAddress with newTokenAddress.
  3. Assign usdtToken to the new token contract instance (IERC20(newTokenAddress)).
  4. Emit the UpdatedUsdtTokenAddress event.
- If tokenType is WBTC:
  1. Verify that newTokenAddress is different from wbtcTokenAddress.
     - If not, revert with: "WBTC token address has already been modified to that value".
  2. Update wbtcTokenAddress with newTokenAddress.
  3. Assign wbtcToken to the new token contract instance (IERC20(newTokenAddress)).
  4. Emit the UpdatedWbtcTokenAddress event.
- If tokenType is WETH:
  1. Verify that newTokenAddress is different from wethTokenAddress.
     - If not, revert with: "WETH token address has already been modified to that value".
  2. Update wethTokenAddress with newTokenAddress.
  3. Assign wethToken to the new token contract instance (IERC20(newTokenAddress)).
  4. Emit the UpdatedWethTokenAddress event.

Step 3: If tokenType is not one of the predefined types, revert with: "Invalid token type".

Post Condition

The address of the specified token (USDC, USDT, WBTC, or WETH) is updated to newTokenAddress.
The corresponding token contract instance is updated.
An appropriate event is emitted to signal the update.

Alternative flows

Zero Address Provided:

- If newTokenAddress is the zero address (address(0)), the transaction reverts with:
  "The token address cannot be the zero address".

Same Address as Current Token:

- If newTokenAddress equals the current address of the specified token, the transaction reverts with a message specific to the token type:
  - "USDC token address has already been modified to that value".
  - "USDT token address has already been modified to that value".
  - "WBTC token address has already been modified to that value".
  - "WETH token address has already been modified to that value".

Invalid Token Type:

- If tokenType is not one of USDC, USDT, WBTC, or WETH, the transaction reverts with:
  "Invalid token type".

Unauthorized Caller:

- If a non-owner address attempts to execute the function, the onlyOwner modifier reverts the transaction.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 21: Update Price Feed Address	
Title	Update the Price Feed Address - updatePriceFeedAddress(TokenType tokenType,address newPriceFeedAddress)
Description	Updates the price feed address for a specific token type while validating that the new address corresponds to the expected price oracle.
Actors and interfaces	• Smart contract owner. • AggregatorV3Interface for interacting with the price oracle.
Initial status and preconditions	• The caller must be the owner of the smart contract. • The provided newPriceFeedAddress must not be the zero address. • The newPriceFeedAddress must not match the current price feed address for the specified token type. • The newPriceFeedAddress must correspond to a valid TOKEN/USD price oracle.
Basic Flow	
Step 1: Verify that the caller is the contract owner (onlyOwner modifier).	
Step 2: Validate that the newPriceFeedAddress is not the zero address.	
Step 3: Initialize a temporary AggregatorV3Interface instance using the provided newPriceFeedAddress.	
Step 4: Determine the expected TOKEN/USD description hash based on the provided tokenType.	
Step 5: Ensure the newPriceFeedAddress is not already the current price feed address for the token type.	
Step 6: Retrieve the description from the temporary price feed and compute its hash.	

Step 7: Compare the computed hash with the expected description hash to ensure validity.
Step 8: Update the appropriate price feed address and interface for the specified tokenType.
Step 9: Emit an event indicating the update of the price feed address.
Post Condition
The price feed address and corresponding AggregatorV3Interface for the specified tokenType are updated. The appropriate update event (Updated<Matic/Usdc/Usdt/Wbtc/Weth>PriceFeedAddress) is emitted.
Alternative flows
Invalid Address: • If newPriceFeedAddress is the zero address, the transaction reverts with the error: *"The price data feed address cannot be the zero address"*. Repeated Update: • If newPriceFeedAddress matches the current price feed address for the tokenType, the transaction reverts with an error like: *"<TOKEN> price feed address has already been modified to that value"*. Invalid Description: • If the description of the new price feed does not match the expected TOKEN/USD format, the transaction reverts with the error: *"The new address does not seem to belong to the correct price data feed"*. Invalid Token Type: • If an unsupported tokenType is provided, the transaction reverts with the error: *"Invalid token type"*. Error in Description Retrieval: • If the description() function call fails, the transaction reverts with the error: *"The new address does not seem to belong to the correct price data feed"*.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 22: Get Current Token Price

Title	Retrieve the Current Token Price - getCurrentTokenPrice(TokenType tokenType)
Description	Fetches the current price of a specified token type from its associated oracle and ensures the validity of the price data.

Actors and interfaces	Any user or external contract interacting with the function. AggregatorV3Interface for retrieving price data from oracles.
Initial status and preconditions	• The tokenType must be a valid token supported by the contract (e.g., MATIC, USDC, USDT, WBTC, WETH). • The oracle associated with the tokenType must be initialized and accessible. • The function caller must ensure they have sufficient computational resources (gas) for the view call.
Basic Flow	
Step 1: Determine the appropriate price feed (AggregatorV3Interface) based on the provided tokenType.	
Step 2: Use the selected price feed to attempt fetching the latest round data (latestRoundData).	
Step 3: Validate the fetched data: • Ensure the price (answer) is positive. • Verify the timeStamp is valid (non-zero and not exceeding the current block timestamp). • Confirm the round data is complete (answeredInRound >= roundID). • Check the price data freshness (block.timestamp - timeStamp <= MAX_PRICE_AGE).	
Step 4: Convert the answer (price) to an unsigned integer and return it.	
Post Condition	
The function returns the current price of the specified token type as a uint256. No changes are made to the contract's state since this is a view function.	
Alternative flows	
Invalid Token Type: • If an unsupported tokenType is provided, the transaction reverts with: *"Invalid token type"*. Oracle Fetch Failure: • If the latestRoundData() function call fails, the transaction reverts with: *"There was an error obtaining the token price from the oracle"*. Invalid Price Data: • If the answer is non-positive, the transaction reverts with: *"Invalid price data from oracle"*. Stale Price Data: • If the timeStamp is zero, exceeds the current block timestamp, or indicates price data older than MAX_PRICE_AGE, the transaction reverts with: *"Stale price data"*. Incomplete Round Data: • If answeredInRound < roundID, the transaction reverts with: *"Incomplete round data"*.	

Executed internally by the smart contract

Function “InvestFromCryptoToken” or “InvestFromMatic” -> Function “Obtain current price of MATIC” -> Consultation with a decentralized oracle about the CryptoToken/USD pair -> Return of value to the smart contract

Use case # 21: Pause	
Title	Pause the Contract - pause()
Description	Pauses all functionalities of the contract, effectively halting its operations. Can only be executed by the owner of the contract.
Actors and interfaces	• Contract owner (only account allowed to execute this function). • Internal pause functionality provided by the _pause() function (likely inherited from a contract such as OpenZeppelin's Pausable contract).
Initial status and preconditions	• The address executing the function must be the owner of the contract (as enforced by the onlyOwner modifier). • The contract must not already be paused; otherwise, the _pause() function will have no effect (this is handled internally and doesn't cause a revert). • The function is typically called by the owner to prevent further contract interactions during critical maintenance, security updates, or other events requiring a pause.
Basic Flow	
Step 1: The function is invoked by the contract owner.	
Step 2: The onlyOwner modifier ensures that only the owner of the contract can call the function.	
Step 3: The _pause() function is executed, halting all contract functions that are protected by the "paused" state (e.g., via modifiers like whenNotPaused).	
Step 4: The contract is now paused, and all paused operations are temporarily suspended until the contract is unpaused.	
Post Condition	
The contract is paused, and no further functions that rely on the "paused" state can be executed until the contract is unpaused by the owner. No changes to the contract state variables other than the paused state are expected.	
Alternative flows	
Non-owner tries to execute the function: • If an address other than the owner attempts to call this function, the transaction is reverted with the error: *"Ownable: caller is not the owner"* (enforced by the onlyOwner modifier).	

Contract already paused: • If the contract is already paused, the _pause() function is called but no additional changes occur; the state remains paused. There is no revert.

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 24: Unpause	
Title	Unpause the Contract - unpause()
Description	Unpauses all functionalities of the contract, allowing it to resume its operations. This function can only be executed by the owner of the contract.
Actors and interfaces	• Contract owner (only account authorized to call this function). • Internal unpause functionality provided by the _unpause() function (likely inherited from a contract such as OpenZeppelin's Pausable contract).
Initial status and preconditions	• The address executing the function must be the owner of the contract (as enforced by the onlyOwner modifier). • The contract must be paused. If it is not paused, the function will still execute, but it will have no effect on the contract state. • The function is used by the owner to resume normal operations after the contract has been paused.
Basic Flow	
Step 1: The function is invoked by the contract owner.	
Step 2: The onlyOwner modifier ensures that only the owner can execute the function.	
Step 3: The _unpause() function is executed, allowing the contract to resume operations.	
Step 4: All functions that were previously paused (protected by the "paused" state) can now be executed again.	
Post Condition	
The contract is no longer paused, and all functionalities previously suspended by the paused state are now active and can be used again. No other state variables are expected to change except for the paused state being set to false.	
Alternative flows	

Non-owner tries to execute the function:

- If an address other than the owner tries to invoke this function, the transaction is reverted with the error: *"Ownable: caller is not the owner"* (enforced by the onlyOwner modifier).

Contract is not paused:

- If the contract is already unpaused, calling this function will have no effect on the contract's state. There will be no revert, but no change in state either.

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 25: Transfer Ownership of the Contract

Title	Transfer Ownership of the Contract - transferOwnership(address newOwner)
Description	Transfers the ownership of the contract to a new address, specified by newOwner. This function can only be called by the current owner and ensures that the new owner is valid (not the zero address or the contract's own address).
Actors and interfaces	• Contract owner (the only account authorized to execute this function). • Inherits from Ownable (typically from OpenZeppelin's Ownable contract). • Internal function _transferOwnership is called to handle the actual transfer of ownership.
Initial status and preconditions	• The address executing the function must be the current owner of the contract (enforced by the onlyOwner modifier). • The newOwner parameter must be a valid address: it cannot be the zero address (0x0000000000000000000000000000000000000000) or the contract's own address (as this would result in a loss of control over the contract). • The function requires gas to be executed.
Basic Flow	
Step 1: The function is called by the current owner, passing the address of the new owner (newOwner).	

Step 2: The function checks if the newOwner is the zero address. If true, the transaction is reverted with the error message *"Ownable: new owner is the zero address"*.
Step 3: The function checks if the newOwner is the contract's own address. If true, the transaction is reverted with the error message *"Ownable: new owner cannot be the same contract address"*.
Step 4: The function proceeds to call _transferOwnership(newOwner), which updates the ownership of the contract to the newOwner.
Post Condition
The ownership of the contract is successfully transferred to the newOwner. The previous owner no longer has control over the contract. The newOwner address is recorded as the new owner of the contract.
Alternative flows
New Owner is the Zero Address: • If the newOwner address is the zero address, the transaction is reverted with the error message *"Ownable: new owner is the zero address"*. New Owner is the Contract Address: • If the newOwner address is the contract's own address, the transaction is reverted with the error message *"Ownable: new owner cannot be the same contract address"*. Caller is Not the Owner: • If the caller is not the current owner, the transaction will fail due to the onlyOwner modifier, reverting with the error message *"Ownable: caller is not the owner"*.

Executed by an owner with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 26: Receive	
Title	Handle Incoming MATIC Transactions - receive()
Description	This function handles incoming MATIC transactions, allowing the contract to receive MATIC. It emits an event (MaticReceived) to signal the received MATIC. This function can only be called by the contract's owner and is protected against reentrancy attacks.
Actors and interfaces	Contract Owner: The only actor allowed to call this function. Sender: The address sending MATIC to the contract.

Initial status and preconditions	• The address executing the function must be the contract owner (as enforced by the onlyOwner modifier). • The function must be called as a result of an incoming MATIC transaction (i.e., the contract is receiving MATIC). • The contract must be deployed and have sufficient balance to accept MATIC. • Gas is required for the execution of the transaction. • The function is protected by the nonReentrant modifier to prevent reentrancy attacks.
Basic Flow	
Step 1: The contract receives MATIC through a transfer from an external account.	
Step 2: The contract checks that the sender is the owner (due to the onlyOwner modifier).	
Step 3: The receive() function is triggered automatically, and the contract records the amount of MATIC sent (msg.value) and the sender's address (msg.sender).	
Step 4: The contract emits the MaticReceived event, including the sender's address and the amount of MATIC received.	
Step 5: The transaction is completed successfully.	
Post Condition	
The contract receives the MATIC sent by the external address. The MaticReceived event is emitted, notifying that MATIC has been received by the contract. The contract's balance increases by the amount of MATIC received.	
Alternative flows	
Caller is Not the Owner: • If the caller is not the contract owner, the onlyOwner modifier will prevent the transaction from proceeding, reverting with the error message *"Ownable: caller is not the owner"*. Non-MATIC Transfer: • This function is specifically for receiving MATIC transactions. If another type of transaction occurs (e.g., tokens), this function will not trigger; the contract would need to handle such transactions in a separate function, such as transfer(). Reentrancy Attack: • The nonReentrant modifier ensures that no reentrancy attacks can occur. If an attack is attempted, the transaction will be reverted with an error message related to reentrancy.	

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front.

Use case # 27: (Proxy Admin) Renounce Ownership

Title	Renounce Ownership - renounceOwnership()
Description	This function allows the current owner to renounce ownership of the contract, transferring ownership to the zero address (address(0)). After renouncing, no functions protected by the onlyOwner modifier can be called, effectively leaving the contract without an owner.
Actors and interfaces	• Contract Owner: The only actor who can call this function to renounce ownership. • The function interacts with the internal ownership management mechanism, specifically transferring ownership to address(0).
Initial status and preconditions	• The contract must have an owner, and the address executing the function must be the current owner. • The address executing the function must have sufficient gas to complete the transaction. • The function is protected by the onlyOwner modifier, which ensures that only the current owner can call it.
Basic Flow	
Step 1: The current owner calls the renounceOwnership() function.	
Step 2: The contract checks if the caller is the current owner using the onlyOwner modifier.	
Step 3: The contract executes the _transferOwnership(address(0)) function, transferring ownership to the zero address.	
Step 4: Ownership is effectively renounced, and the contract is left without an owner.	
Step 5: The function completes successfully, and the contract no longer has an owner.	
Post Condition	
• The contract no longer has an owner, as ownership has been transferred to address(0). • Any functions or actions restricted to the owner (using the onlyOwner modifier) are now inaccessible. • The state of the contract is such that it is effectively ownerless and cannot be managed by a specific address.	
Alternative flows	
Caller is Not the Owner: • If the caller is not the contract owner, the onlyOwner modifier will prevent the transaction from proceeding, and it will revert with the error message *"Ownable: caller is not the owner"*. Invalid Contract State:	

- If there is any issue preventing the _transferOwnership() function from executing properly (e.g., an internal failure), the transaction will revert and not renounce ownership.

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 28: (Proxy Admin) Transfer Ownership	
Title	Transfer Ownership - transferOwnership(address newOwner)
Description	This function transfers ownership of the contract to a new address (newOwner). It updates the _owner state variable and emits an OwnershipTransferred event to signal the change in ownership. The function is protected by the onlyOwner modifier, which ensures that only the current owner can call it.
Actors and interfaces	The function interacts with the internal _owner state variable, which holds the address of the current contract owner.
Initial status and preconditions	• The contract must have an existing owner (a valid address is stored in the _owner variable). • The function does not require any specific parameters to be met beyond the valid newOwner address being passed to it by the calling function.
Basic Flow	
Step 1: The newOwner address is passed as a parameter to the function.	
Step 2: The function stores the current _owner in the variable oldOwner.	
Step 3: The _owner state variable is updated to the newOwner address.	
Step 4: The function emits an OwnershipTransferred event, passing the oldOwner and newOwner addresses.	
Step 5: The ownership is successfully transferred, and the function execution completes.	
Post Condition	
• The _owner state variable is updated to reflect the newOwner address. • The contract's ownership has been transferred to the new address. • An OwnershipTransferred event is emitted, notifying the change of ownership.	

Alternative flows
Invalid Address for newOwner: • If the function is called with an invalid or inappropriate address for newOwner, the transaction may fail (although the function itself doesn't enforce validation checks). If validation logic is implemented externally, such as in transferOwnership, the transaction may revert before _transferOwnership is invoked. No Change in Ownership: • If the newOwner address is the same as the current _owner, the function will still emit the OwnershipTransferred event, but the ownership state will remain unchanged. If additional checks are added (e.g., ensuring the newOwner is not the same as the current owner), the transaction might be reverted.

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 29: (Proxy Admin) Upgrade and Call	
Title	Upgrade and Call - upgradeAndCall(ITransparentUpgradeableProxy proxy, address implementation, bytes memory data)
Description	The function upgrades the specified proxy contract to a new implementation and optionally calls a function on the upgraded implementation. It requires the contract to be the admin of the proxy. If no data is provided, no ETH is sent with the transaction.
Actors and interfaces	• Contract Owner: The owner is the only actor who can call this function as it is restricted by the onlyOwner modifier. • ITransparentUpgradeableProxy: This is the interface for the proxy contract that will be upgraded. • Implementation Contract: The contract that will replace the current implementation in the proxy.
Initial status and preconditions	• The contract executing this function must be the **admin** of the proxy contract (i.e., it must have the required permissions to upgrade the proxy).

	• The proxy contract (proxy) must be deployed and must be an instance of the ITransparentUpgradeableProxy interface. • The address provided for the implementation must be a valid contract address. • If no data (data) is passed, the transaction must not include any value (msg.value must be zero). If data is provided, msg.value can be any value, depending on the needs of the function called on the new implementation.
Basic Flow	
Step 1: The contract owner calls upgradeAndCall, passing the proxy contract address (proxy), the new implementation contract address (implementation), and the data (data) to execute on the upgraded implementation.	
Step 2: The contract checks that the sender is the contract owner (via the onlyOwner modifier).	
Step 3: The function calls the upgradeToAndCall method on the proxy contract, passing the implementation address and the data.	
Step 4: If data is not empty, the function call specified in data is executed on the new implementation contract. If data is empty, no function call is executed, but the contract is upgraded.	
Step 5: If msg.value is provided, it is forwarded to the proxy.upgradeToAndCall function.	
Step 6: The proxy contract is now upgraded to the new implementation, and any additional function call (if provided) is executed.	
Post Condition	
• The proxy contract is upgraded to the new implementation (implementation). • If data was provided, the specified function on the new implementation is executed. • The proxy is now using the upgraded logic provided by the new implementation contract. • The state of the contract (in terms of the proxy's logic) has changed, and any funds (msg.value) sent with the transaction (if applicable) are handled by the proxy contract.	
Alternative flows	
Proxy Admin Validation: • If the contract calling upgradeAndCall is not the admin of the proxy, the transaction is reverted with an error indicating insufficient permissions. Empty data with Non-Zero msg.value: • If the data is empty but msg.value is non-zero, the transaction may revert if it violates the condition requiring msg.value to be zero when data is empty. Invalid Implementation Address:	

- If the implementation address is invalid or the contract at the implementation address does not implement the expected logic, the transaction may fail, and the upgrade operation will not occur.

Failure in Function Call on Implementation:

- If the data is provided and the function call specified in data fails (e.g., due to invalid parameters or a reverted transaction), the entire transaction will be reverted.

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Smart Contract implementation process

1. The process starts using a code editor such as **Visual Studio code** in which the smart contract will be programmed using the **Solidity** programming language.

2. Subsequently, the functional, security and gas tests of the smart contract are generated using the **JavaScript programming language,** (Security testing is not a substitute for security audits.)

3. **Yarn** is used to manage the dependencies required by the smart contract, while **Hardhat** is used as a development environment to compile, debug and deploy the smart contract in the testnet.

4. It is possible that to make the development of the smart contract more efficient, Scaffold-ETH2 will be used. At the same time, **Gitlab** is used to perform code version control.

5. With the aim of realistically testing the smart contract, it will be deployed on the **Mumbai testnet network** , which allows you to interact with the smart contract from a blockchain explorer such as **Polygon Scan .**

6. Finally, to deploy the smart contract in production, the **Polygon Blockchain** and **hardhat** will be used to carry out the deployment.

7. Once the smart contract is uploaded to the blockchain, it will be possible to interact with it using the **ethers.js** library and a provider like **metamask** , using **ethers.js** and the **Alchemy API** or using a crypto asset wallet from a blockchain explorer as in the case of **Polygon Scan** .

References

- Smart contract programming language: Solidity version +0.8.0
  Documentation: https://docs.soliditylang.org/en/v0.8.21/

- Smart contract testing programming language: Javascript latest version

Documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript

- Code editor: Visual studio code
  Documentation: https://code.visualstudio.com/docs

- Development environment to compile, debug and deploy smart contracts : Hardhat
  Documentation: https://hardhat.org/docs

- Package manager to manage dependencies in the project: Yarn
  Documentation: https://yarnpkg.com/getting-started

- Toolkit to make smart contract development more efficient: Scaffold-ETH2
  Documentation: https://scaffold-eth-2-docs.vercel.app/

- Version control systems for source code management in software development: Github / Gitlab
  Documentation: https://docs.gitlab.com/

- Blockchain explorer to access and interact with smart contracts : Polygon scan
  Documentation: https://docs.Polygonscan.com/

- Blockchain for testing replica of the Polygon blockchain: Mumbai testnet
  Documentation: https://forum.polygon.technology/t/introducing-the-amoy-testnet-for-polygon-pos/13388

- Blockchain to deploy smart contracts in production: Polygon blockchain
  Documentation: https://www.coinbase.com/es/learn/crypto-basics/what-is-Polygon

    - Crypto token addresses over polygon testnet (amoy)
        - POL Native token. Must be obtained from Faucet https://faucet.polygon.technology/
        - https://www.alchemy.com/faucets/polygon-amoy
        - USDC https://amoy.polygonscan.com/address/0xF68054bFe5D45432ffCA28fFA1F3D685d0456Ddc
        - USDT https://amoy.polygonscan.com/address/0x70e02Fb82B6BC04F64099689B0599e14B44D4fBb

    - WBTC https://amoy.polygonscan.com/address/0x3C8df3C48B3884DA2ff25e17524282d60F9C3b93
    - WETH https://amoy.polygonscan.com/address/0x524a89ED77d5827320E35E12bCA96830C6b7960A

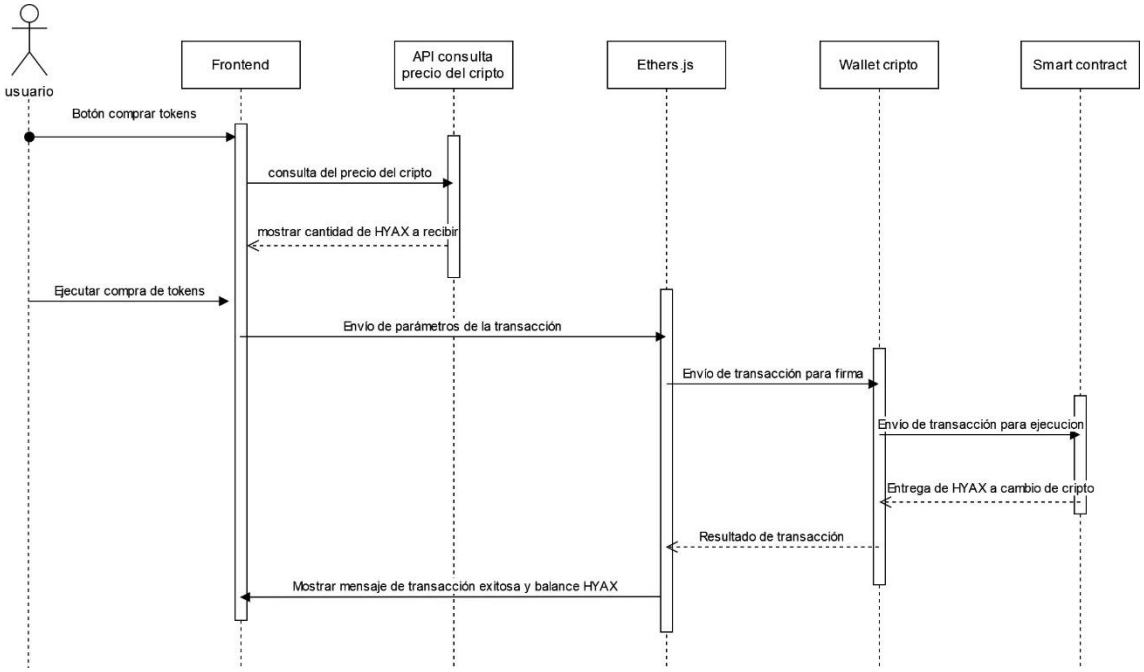
- Crypto token addresses on ethereum testnet (sepolia)

    - MATIC/ETH Native token. Must be obtained from Faucet https://sepoliafaucet.com/
    - USDC https://sepolia.etherscan.io/address/0x256452E79137B1D8f3c7eb3Ed10d9eb782F43BE1#code
    - USDT https://sepolia.etherscan.io/address/0xfd070C28BD649624080637A16F994161B6Fb84e1#code
    - WBTC https://sepolia.etherscan.io/address/0x516fd969524C6f0f429ffca1959521610c0364D7#code
    - WETH https://sepolia.etherscan.io/address/0x8CdC4fc4e4C717b32CbDcA6f0c80093e8bCC071C#code

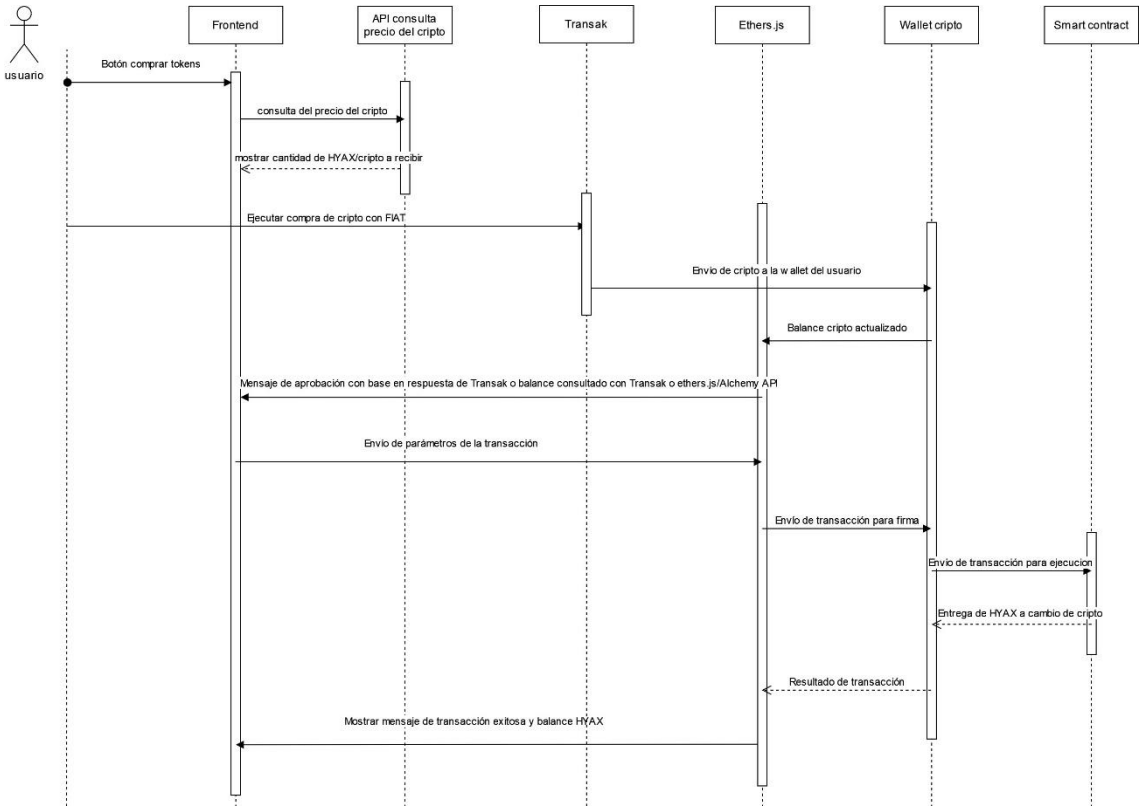
- Crypto token addresses on polygon mainnet
    - MATIC/ETH Native token
    - USDC 0x3c499c542cef5e3811e1192ce70d8cc03d5c3359
    - USDT 0xc2132d05d31c914a87c6611c10748aeb04b58e8f
    - WBTC 0x1bfd67037b42cf73acf2047067bd4f2c47d9bfd6
    - WETH 0x7ceb23fd6bc0add59e62ac25578270cff1b9f619

- Oracle price data powered by chainlink over polygon mainnet
    - MATIC / USD 0xAB594600376Ec9fD91F8e885dADF0CE036862dE0
    - USDC / USD 0xfE4A8cc5b5B2366C1B58Bea3858e81843581b2F7
    - USDT / USD 0x0A6513e40db6EB1b165753AD52E80663aeA50545
    - WBTC / USD 0xDE31F8bFBD8c84b5360CFACCa3539B938dd78ae6
    - WETH / USD 0xF9680D99D6C9589e2a93a78A04A279e509205945

HYAX token purchase sequence diagram - Crypto



HYAX token purchase sequence diagram - Transak


usuario
Frontend
API consulta precio del cripto
Transak
Ethers.js
Wallet cripto
Smart contract
Botón comprar tokens
consulta del precio del cripto
mostrar cantidad de HYAX/cripto a recibir
Ejecutar compra de cripto con FIAT
Envio de cripto a la w allet del usuario
Balance cripto actualizado
Mensaje de aprobación con base en respuesta de Transak o balance consultado con Transak o ethers.js/Alchemy API
Envio de parámetros de la transacción
Envio de transacción para firma
Envio de transacción para ejecucion
Entrega de HYAX a cambio de cripto
Resultado de transacción
Mostrar mensaje de transacción exitosa y balance HYAX