smart CONTRACT & BLOCKCHAIN DOCUMENTATION

Base specifications of the smart contract – PART II

**HYDRAXIS SMART CONTRACT PART II**

Sep 2024

Contents

[Use case # 1: Constructor 2](#_Toc176770957)

[Use case # 2: Add wallet to whitelist 2](#_Toc176770958)

[Use case # 3: Remove wallet from whitelist 2](#_Toc176770959)

[Use case # 4 Fund Smart Contract 3](#_Toc176770960)

[Use case # 5: Withdraw Locked Tokens for Team Member 3](#_Toc176770961)

[Use case # 6: Withdraw Locked Tokens For Growth 3](#_Toc176770962)

[Use case # 5: Update Values of Token Rewards 3](#_Toc176770963)

[Use case # 6: Withdraw Token Rewards 4](#_Toc176770964)

[Use case # 7: Withdraw Tokens to Burn 4](#_Toc176770965)

[Use case # 8: Update waiting time 4](#_Toc176770966)

[(EXTERNAL) Use case # 9: Calculate Token Rewards Earned Weekly 4](#_Toc176770967)

[Test Driven Design Plan 6](#_Toc176770968)

[Testing Use Case #1: Constructor 6](#_Toc176770969)

[Testing Use Case #2: Add Wallet to Whitelist 8](#_Toc176770970)

[Testing Use Case #3: Remove Wallet from Whitelist 10](#_Toc176770971)

[Use Case #4: Fund Smart Contract 12](#_Toc176770972)

[Use Case #5: Withdraw Locked Tokens for Team Member 14](#_Toc176770973)

[Use Case #6: Withdraw Locked Tokens for Growth 16](#_Toc176770974)

[Use Case #7: Update Values of Token Rewards 18](#_Toc176770975)

[Use Case #8: Withdraw Token Rewards 20](#_Toc176770976)

[Use Case #9: Withdraw Tokens to Burn 22](#_Toc176770977)

[Use Case #10: Update Waiting Time 24](#_Toc176770978)

[External Use Case #11: Calculate Token Rewards Earned Weekly 26](#_Toc176770979)

|  |  |
| --- | --- |
| Use case # 1: Constructor | |
| Title | Constructor - constructor( ) |
| Description | Functionality that allows an address on the Polygon network to deploy the smart contract with the required initial parameters |

|  |  |
| --- | --- |
| Use case # 2: Add wallet to whitelist | |
| Title | addWalletToWhitelist(WalletType {TeamMember, Investor}, address investorAddress) |
| Description | Functionality exclusive to the smart contract owner or the whitelister.  A new wallet gets added to the list of wallets and gets whitelisted as TeamMember or Investor.  An struct gets created to store the WalletType, amount of HYAX locked (in case of team member), holding time, accumulated rewards and bitcoin reward address.  It’s possible for the smart contract owner to add wallets and their amount of HYAX locked in the future based on new investors that join the company.  Each team member has the ability to see his own token balance in the smart contract. |

|  |  |
| --- | --- |
| Use case # 3: Remove wallet from whitelist | |
| Title | removeWalletToWhitelist(address investorAddress) |
| Description | Functionality exclusive to the smart contract owner or the whitelister.  A wallet address is sent to the function by the owner.  The smart contract changes the state of the isWhitelisted  property to “false” the address and it’s corresponding Wallet Data from the whitelist. |

|  |  |
| --- | --- |
| Use case # 4 Fund Smart Contract | |
| Title | fundSmartContract(FundingType {TeamRewards, InvestorRewards, GrowthTokens}, uint256 amount) |
| Description | Functionality exclusive to the smart contract owner.  Allows the smart contract owner to send HYAX tokens to the smart contract that will be used in investor rewards, team rewards or growth tokens. |

|  |  |
| --- | --- |
| Use case # 5: Withdraw Team Tokens | |
| Title | withdrawTeamTokens() |
| Description | Functionality exclusive to the members of the team that have been whitelisted by the owner.  It verifies if the waiting time has totally elapsed in order to do a token withdrawal.  It’s only possible to withdraw the first 20% the exact same day after 4 years (48 months) elapsed since the tokens were initially locked in,  The following 20% withdraw will be only possible after 12 months have elapsed and it will repeat for a total of 5 times, until the 100% of tokens have been withdrawn. |

|  |  |
| --- | --- |
| Use case # 6: Withdraw Growth Tokens | |
| Title | withdrawGrowthTokens() |
| Description | Functionality exclusive to the owner of the smart contract.  The smart contract can only allow to withdraw to the owner the deposited tokens at a rate of 5% per year for 20 years.  The first 5% will be possible to withdraw after exactly 1 year (12 months) of depositing the tokens. |

|  |  |
| --- | --- |
| Use case # 5: Update Values of Token Rewards | |
| Title | updateValuesOfTokenRewards(mapping rewards (address investorAddress –> uint256 rewardAmount)) |
| Description | Functionality exclusive to the owner of the smart contract or the rewards updater address.  It allows to update the information of the token rewards earned by each user during the week.  The update is based on the calculation made externally by an automated script created using javascript/typescript (See use case #9). |

|  |  |
| --- | --- |
| Use case # 6: Withdraw Token Rewards | |
| Title | withdrawTokenRewards( ) |
| Description | Functionality exclusive to users in the whitelist (team members and investors).  It allows users (team members and investors whitelisted in the Hydraxis platform), to retire token rewards based on the number of tokens they have earned on a weekly basis.  After withdrawing the tokens, it updates the accumulated rewards balance to zero. |

|  |  |
| --- | --- |
| Use case # 7: Withdraw Tokens to Burn | |
| Title | withdrawTokensToBurn(uint256 amount) |
| Description | Functionality exclusive to the owner of the smart contract.  It allows the owner to withdraw tokens of the smart contract to burn them, in case users don’t withdraw them after a year of finishing the token rewards. |

|  |  |
| --- | --- |
| Use case # 8: Update waiting time | |
| Title | updateWaitingTime(uint8 FundingType, uint256 amount) |
| Description | Functionality exclusive to the owner of the smart contract.  Allows the investor to update the waiting time to withdraw tokens for team rewards, investor rewards and growth tokens. |

|  |  |
| --- | --- |
| (EXTERNAL) Use case # 9: Calculate Token Rewards Earned Weekly | |
| Title | calculateTokenRewardsEarnedWeekly() |
| Description | Functionality external to the smart contract. It’s a script made using javascript and it’s hosted in a secure server. The same way as the whitelister script and address was hosted in the PART I of the smart contract.  It calculates the rewards earned by each user address based on a snapshot of specific days of the week.  Rewards to offer are calculated based on the weekly number of tokens that are available to offer to people who hold the token.  It sends the value of the earned rewards to the smart contract on a weekly basis to update the balance of users in the smart contract, so they can see it in real time.  Rewards for team members are calculated based on the number of tokens they have locked initially in the smart contract (input made when the wallet was added to the whitelist.), divided by the total tokens of team members in the whitelist. Multiplied by the total token rewards for team members per week.  Rewards for investors are calculated based on the number of tokens they have holding in their wallet (snapshot of the week), divided by the total tokens of investors that approved Hydraxis KYC. Multiplied by the total token rewards for investors per week. |

|  |  |
| --- | --- |
| Use case # 10: Pause smart contract | |
| Title | Pause smart contract - pause( ) |
| Description | Functionality that allows the owner address of the smart contract to stop exchanges and transfers of HYAX tokens over the Polygon network |
| Actors and interfaces | Owner address of the smart contract |
| Initial status and preconditions | The smart contract is not paused  The owner's crypto address has MATIC to pay the transaction gas. |
| Basic Flow | |
| Step 1: The owner address executes the pause function | |
| Step 2: The smart contract validates that it is the address of the owner who executes the function | |
| Step 3: The smart contract validates that the smart contract is not paused | |
| Post Condition | |
| Smart contract stops token exchanges and transfers | |
| Alternative flows | |
| If the address does not have enough MATIC to pay for the transaction gas, the transaction is reverted.  If the executor of the function is not the owner of the smart contract, the transaction is reverted. | |

|  |  |
| --- | --- |
| Use case # 11: Unpause smart contract | |
| Title | Unpause from smart contract - unpause ( ) |
| Description | Functionality that allows the owner address of the smart contract to re-activate exchanges and transfers of HYAX tokens over the Polygon network |
| Actors and interfaces | Owner address of the smart contract |
| Initial status and preconditions | The smart contract is currently paused  The owner's crypto address has MATIC to pay the transaction gas. |
| Basic Flow | |
| Step 1: The owner address executes the unpause function | |
| Step 2: The smart contract validates that it is the address of the owner who executes the function | |
| Step 3: The smart contract validates that the smart contract is currently paused | |
| Post Condition | |
| The smart contract re-activates token exchanges and transfers | |
| Alternative flows | |
| If the address does not have enough MATIC to pay for the transaction gas, the transaction is reverted.  If the executor of the function is not the owner of the smart contract, the transaction is reverted. | |

|  |  |
| --- | --- |
| Use case # 12: Change smart contract owner | |
| Title | Change owner of smart contract - transferOwnership (address \_newOwner) |
| Description | Functionality that allows the smart owner address to transfer ownership of the smart contract to another address on the Polygon network |
| Actors and interfaces | Owner address of the smart contract |
| Initial status and preconditions | The crypto address that will execute this function is currently registered as the owner of the smart contract.  The crypto address of the current owner has MATIC to pay the transaction gas. |
| Basic Flow | |
| Step 1: The address of the current owner, enter as a parameter the address that will be the new owner and execute the functionality of transferring ownership of the smart contract | |
| Step 2: The smart contract validates that it is the address of the owner who executes the function | |
| Step 3: The smart contract validates that the new owner address of the Smart contract is not invalid (zero address) | |
| Step 4: The smart contract validates that the new owner address of the Smart contract is not the same as the address of the Smart contract | |
| Post Condition | |
| The owner address of the smart contract is updated with the address of the new owner entered by parameter | |
| Alternative flows | |
| If the address does not have enough MATIC to pay for the transaction gas, the transaction is reverted.  In case an invalid address is entered in the function, the transaction is reverted.  If the executor of the function is not the owner of the smart contract, the transaction is reverted. | |

|  |  |
| --- | --- |
| Use case # 13: Update white lister address | |
| Title | Update white lister address - updateWhiteListerAddres (address \_ newWhiteListerAddress) |
| Description | Functionality that gives the owner of the smart contract the ability to update the address that will be responsible for modifying the white list of addresses that can interact with the smart contract |
| Actors and interfaces | Smart contract owner |
| Initial status and preconditions | A white list address is already defined in the smart contract.  The owner's address has MATIC to pay the gas for the transaction. |
| Basic Flow | |
| Step 1: The owner of the smart contract enters by parameter the new address that will have the white list capabilities, and sends the transaction | |
| Step 2: The smart contract validates that the sender of the transaction is the owner | |
| Step 3: The smart contract validates that the white lister new address parameter is not an invalid value (zero address) | |
| Step 4: The smart contract validates that the new white lister address parameter is not the same as the one currently registered | |
| Post Condition | |
| The smart contract updates the white list address.  UpdatedWhiteListerAddress event is emitted with the new white lister address. | |
| Alternative flows | |
| If the address does not have enough MATIC to pay for the transaction gas, the transaction is reverted.  In case an invalid address is entered in the function, the transaction is reverted.  In the event that a white list address equal to the one currently registered is entered, the transaction is reverted.   If the sender of the transaction is not the owner, the transaction is reverted. | |

|  |  |
| --- | --- |
| Use case # 14: Update rewards updater address | |
| Title | Update rewards updater address - updateRewardsUpdater Address (address \_ newRewardsUpdaterAddress) |
| Description | Functionality that gives the owner of the smart contract the ability to update the address that will be responsible for modifying the rewards of the addresses whitelister in the smart contract |
| Actors and interfaces | Smart contract owner |
| Initial status and preconditions | A rewards updated address is already defined in the smart contract.  The owner's address has MATIC to pay the gas for the transaction. |
| Basic Flow | |
| Step 1: The owner of the smart contract enters by parameter the new address that will have the rewards updater capabilities, and sends the transaction | |
| Step 2: The smart contract validates that the sender of the transaction is the owner | |
| Step 3: The smart contract validates that the white lister new address parameter is not an invalid value (zero address) | |
| Step 4: The smart contract validates that the new rewards updater address parameter is not the same as the one currently registered | |
| Post Condition | |
| The smart contract updates the white list address.  UpdatedRewardsUpdaterAddress event is emitted with the new white lister address. | |
| Alternative flows | |
| If the address does not have enough MATIC to pay for the transaction gas, the transaction is reverted.  In case an invalid address is entered in the function, the transaction is reverted.  In the event that a rewards updater address equal to the one currently registered is entered, the transaction is reverted.   If the sender of the transaction is not the owner, the transaction is reverted. | |

# Test Driven Design Plan

# Testing Use Case #1: Constructor

**Test**:

* + Verify that the contract is deployed correctly with the required initial parameters.
  + Ensure that the initial state of the contract is set properly.

**Implementation**:

* Implement the constructor in Solidity to set initial parameters.

**Extended unit tests**

**1. Test Case 1: Successful Deployment with Correct Parameters**

* **Description**: Verify that the contract is deployed correctly with the specified initial parameters.
* **Steps**:
  1. Deploy the contract using the constructor with valid initial parameters.
  2. Check the contract address and ensure it is deployed on the Polygon network.
  3. Verify that the initial parameters are set correctly by accessing them through public getter functions.
* **Expected Result**: The contract should be deployed successfully, and the initial parameters should match the provided values.

**2. Test Case 2: Initial State Verification**

* **Description**: Ensure that the initial state of the contract is set correctly after deployment.
* **Steps**:
  1. Deploy the contract with initial parameters.
  2. Retrieve and verify the initial state variables using public getter functions.
* **Expected Result**: The state variables should be initialized to the values specified in the constructor.

**3. Test Case 3: Constructor Parameters Validation**

* **Description**: Verify that the contract rejects invalid constructor parameters and does not deploy.
* **Steps**:
  1. Attempt to deploy the contract with invalid or out-of-range parameters.
  2. Check for any errors or rejections during deployment.
* **Expected Result**: The contract deployment should fail with appropriate error messages indicating invalid parameters.

# Testing Use Case #2: Add Wallet to Whitelist

**Test**:

* + Verify that only the contract owner can add wallets.
  + Check that the wallet is added with the correct type and data.
  + Ensure the struct is correctly updated with the wallet’s data.

**Implementation**:

* Implement addWalletToWhitelist to update the whitelist mapping and struct.

**Full unit tests:**

1. **Test Case 1: Successful Addition of Wallet**
   * **Description**: Ensure that the owner can successfully add a wallet to the whitelist with the correct type and data.
   * **Steps**:
     1. Call addWalletToWhitelist with valid parameters.
     2. Verify that the wallet is added to the whitelist.
   * **Expected Result**: The wallet should be in the whitelist with the correct type and data.
2. **Test Case 2: Only Owner Can Add Wallet**
   * **Description**: Verify that only the contract owner can add a wallet to the whitelist.
   * **Steps**:
     1. Attempt to call addWalletToWhitelist from a non-owner account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Adding Duplicate Wallet**
   * **Description**: Ensure that adding a wallet that already exists does not create duplicates.
   * **Steps**:
     1. Add a wallet to the whitelist.
     2. Attempt to add the same wallet again with the same or different data.
     3. Verify that the data for the wallet is updated rather than duplicated.
   * **Expected Result**: The wallet data should be updated, not duplicated.
4. **Test Case 4: Adding Wallet with Invalid Data**
   * **Description**: Ensure that adding a wallet with invalid data (e.g., incorrect type or amount) fails.
   * **Steps**:
     1. Call addWalletToWhitelist with invalid parameters.
   * **Expected Result**: The transaction should revert with an error indicating invalid data.
5. **Test Case 5: Validate Wallet Data**
   * **Description**: Verify that the data stored for the wallet is correct.
   * **Steps**:
     1. Add a wallet with specific data.
     2. Retrieve and validate the data stored for the wallet.
   * **Expected Result**: The retrieved data should match the data provided during addition.

# Testing Use Case #3: Remove Wallet from Whitelist

**Test**:

* + Verify that only the contract owner can remove a wallet.
  + Ensure the wallet and its data are removed from the whitelist.

**Implementation**:

* Implement removeWalletFromWhitelist to delete the wallet data from the whitelist.

**Full unit tests:**

1. **Test Case 1: Successful Removal of Wallet**
   * **Description**: Ensure that the owner can successfully remove a wallet from the whitelist.
   * **Steps**:
     1. Add a wallet to the whitelist.
     2. Call removeWalletFromWhitelist for the added wallet.
     3. Verify that the wallet is removed from the whitelist.
   * **Expected Result**: The wallet should be removed and no longer present in the whitelist.
2. **Test Case 2: Only Owner Can Remove Wallet**
   * **Description**: Verify that only the contract owner can remove a wallet from the whitelist.
   * **Steps**:
     1. Attempt to call removeWalletFromWhitelist from a non-owner account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Removing Non-Existent Wallet**
   * **Description**: Ensure that removing a wallet that is not in the whitelist does not cause errors.
   * **Steps**:
     1. Call removeWalletFromWhitelist for a wallet that does not exist in the whitelist.
   * **Expected Result**: The transaction should succeed without altering the whitelist.
4. **Test Case 4: Verify Data Removal**
   * **Description**: Validate that the data associated with a removed wallet is also deleted.
   * **Steps**:
     1. Add a wallet with specific data.
     2. Remove the wallet from the whitelist.
     3. Retrieve data for the removed wallet.
   * **Expected Result**: No data should be retrieved for the removed wallet.
5. **Test Case 5: Re-add Removed Wallet**
   * **Description**: Verify that a wallet can be re-added to the whitelist after being removed.
   * **Steps**:
     1. Add a wallet to the whitelist.
     2. Remove the wallet from the whitelist.
     3. Re-add the wallet to the whitelist.
   * **Expected Result**: The wallet should be successfully re-added with the specified data.

# Testing Use Case #4: Fund Smart Contract

**Test**:

* + Ensure that only the contract owner can fund the contract.
  + Verify that the correct amount and type of funds are added.

**Implementation**:

* Implement fundSmartContract to handle different funding types and update balances.

**Full unit tests:**

1. **Test Case 1: Successful Funding**
   * **Description**: Verify that the owner can successfully fund the contract with a specified amount and type.
   * **Steps**:
     1. Call fundSmartContract with valid parameters.
     2. Verify the contract's balance for the specified funding type.
   * **Expected Result**: The contract's balance should be updated with the funded amount.
2. **Test Case 2: Only Owner Can Fund Contract**
   * **Description**: Ensure that only the contract owner can fund the contract.
   * **Steps**:
     1. Attempt to call fundSmartContract from a non-owner account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Fund with Invalid Amount**
   * **Description**: Ensure that funding with an invalid amount (e.g., zero or negative value) fails.
   * **Steps**:
     1. Call fundSmartContract with an invalid amount.
   * **Expected Result**: The transaction should revert with an error indicating an invalid amount.
4. **Test Case 4: Verify Fund Allocation**
   * **Description**: Validate that funds are correctly allocated based on the specified funding type.
   * **Steps**:
     1. Fund the contract with different types (e.g., TeamRewards, InvestorRewards).
     2. Verify that the allocation reflects the correct type and amount.
   * **Expected Result**: The contract’s balance should reflect the correct allocation for each type.
5. **Test Case 5: Overfunding**
   * **Description**: Verify the contract’s behavior when overfunding with excess tokens.
   * **Steps**:
     1. Fund the contract with an excessive amount.
     2. Verify that the contract handles or rejects the excess funding appropriately.
   * **Expected Result**: The contract should handle or reject excess funding based on predefined limits.

# Testing Use Case #5: Withdraw Locked Tokens for Team Member

**Test**:

* + Ensure only whitelisted team members can withdraw tokens.
  + Verify that withdrawal is allowed based on the time constraints.
  + Check that the correct percentage of tokens is withdrawn.

**Implementation**:

* Implement withdrawLockedTokensForTeamMember with logic to check time constraints and withdrawal percentages.

**Full unit tests:**

1. **Test Case 1: Successful Token Withdrawal**
   * **Description**: Ensure a whitelisted team member can withdraw tokens according to the specified schedule.
   * **Steps**:
     1. Add a team member with locked tokens.
     2. Wait for the required time period.
     3. Call withdrawLockedTokensForTeamMember.
     4. Verify the withdrawn amount and remaining balance.
   * **Expected Result**: Tokens should be withdrawn according to the schedule, and the remaining balance should be updated.
2. **Test Case 2: Withdrawal Timing Constraint**
   * **Description**: Verify that withdrawal is only allowed after the specified time has elapsed.
   * **Steps**:
     1. Add a team member with locked tokens.
     2. Call withdrawLockedTokensForTeamMember before the waiting period elapses.
   * **Expected Result**: The transaction should revert with an error indicating that the waiting time has not yet elapsed.
3. **Test Case 3: Partial Withdrawals**
   * **Description**: Ensure that partial withdrawals are correctly handled (e.g., first 20% on specific dates).
   * **Steps**:
     1. Add a team member with locked tokens.
     2. Perform partial withdrawals on the specified dates.
     3. Verify the withdrawn amounts and remaining balance.
   * **Expected Result**: Withdrawals should follow the specified partial withdrawal schedule.
4. **Test Case 4: Non-Whitelisted Wallet Withdrawal**
   * **Description**: Verify that non-whitelisted addresses cannot withdraw tokens.
   * **Steps**:
     1. Attempt to call withdrawLockedTokensForTeamMember from a non-whitelisted address.
   * **Expected Result**: The transaction should revert with an authorization error.
5. **Test Case 5: Withdrawal After Full Lock Period**
   * **Description**: Ensure that after the full lock period, the remaining tokens can be fully withdrawn.
   * **Steps**:
     1. Add a team member and lock tokens for the full period.
     2. Wait for the full lock period to elapse.
     3. Call withdrawLockedTokensForTeamMember and withdraw the remaining tokens.
   * **Expected Result**: All remaining tokens should be withdrawn successfully.

# Testing Use Case #6: Withdraw Locked Tokens for Growth

**Test**:

* + Ensure only the contract owner can withdraw growth tokens.
  + Verify the correct percentage and timing of withdrawals.

**Implementation**:

* Implement withdrawLockedTokensForGrowth to manage withdrawals based on the specified rate.

**Full unit tests:**

1. **Test Case 1: Successful Growth Token Withdrawal**
   * **Description**: Verify that the contract owner can withdraw growth tokens according to the specified rate.
   * **Steps**:
     1. Fund the contract with growth tokens.
     2. Call withdrawLockedTokensForGrowth.
     3. Verify the withdrawn amount and remaining balance.
   * **Expected Result**: Tokens should be withdrawn at the specified rate, and the remaining balance should be updated.
2. **Test Case 2: Withdrawal Rate Constraint**
   * **Description**: Ensure that the withdrawal rate is correctly enforced (e.g., 5% per year).
   * **Steps**:
     1. Fund the contract with growth tokens.
     2. Attempt to withdraw more than the allowed rate.
   * **Expected Result**: The transaction should revert with an error indicating that the withdrawal exceeds the allowed rate.
3. **Test Case 3: Early Withdrawal Attempt**
   * **Description**: Verify that attempting to withdraw before the specified time (e.g., one year) fails.
   * **Steps**:
     1. Fund the contract with growth tokens.
     2. Attempt to call withdrawLockedTokensForGrowth before the one-year period has elapsed.
   * **Expected Result**: The transaction should revert with an error indicating that the withdrawal is not yet permitted.
4. **Test Case 4: Token Balance Validation**
   * **Description**: Ensure that the token balance of the contract is correctly updated after a withdrawal.
   * **Steps**:
     1. Fund the contract with a specific amount of growth tokens.
     2. Withdraw tokens and check the updated contract balance.
   * **Expected Result**: The contract balance should reflect the withdrawn amount.
5. **Test Case 5: Handling Excess Tokens**
   * **Description**: Verify the contract's behavior when trying to withdraw more tokens than are available.
   * **Steps**:
     1. Fund the contract with a specific amount of growth tokens.
     2. Attempt to withdraw more tokens than are available based on the rate.
   * **Expected Result**: The transaction should revert with an error indicating insufficient tokens for the withdrawal.

# Testing Use Case #7: Update Values of Token Rewards

**Test**:

* + Verify that only the owner or auto updater can update token rewards.
  + Ensure the rewards are updated correctly for each address.

**Implementation**:

* Implement updateValuesOfTokenRewards to update rewards based on external calculations.

**Full unit tests:**

1. **Test Case 1: Successful Update of Rewards**
   * **Description**: Verify that the rewards for each wallet can be successfully updated by the owner or auto updater.
   * **Steps**:
     1. Call updateValuesOfTokenRewards with valid parameters.
     2. Verify that the rewards for each wallet are updated correctly.
   * **Expected Result**: The rewards should be updated as specified.
2. **Test Case 2: Only Authorized Accounts Can Update**
   * **Description**: Ensure that only the owner or auto updater can update the rewards.
   * **Steps**:
     1. Attempt to call updateValuesOfTokenRewards from a non-authorized account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Update with Invalid Data**
   * **Description**: Ensure that updating rewards with invalid data (e.g., negative values) fails.
   * **Steps**:
     1. Call updateValuesOfTokenRewards with invalid reward amounts.
   * **Expected Result**: The transaction should revert with an error indicating invalid data.
4. **Test Case 4: Verify Rewards Update**
   * **Description**: Validate that the rewards data for each wallet reflects the update correctly.
   * **Steps**:
     1. Update rewards for specific wallets.
     2. Retrieve and verify the updated rewards data for those wallets.
   * **Expected Result**: The retrieved data should match the updated values.
5. **Test Case 5: Handling Large Updates**
   * **Description**: Verify the contract’s ability to handle a large number of reward updates.
   * **Steps**:
     1. Call updateValuesOfTokenRewards with a large dataset of reward values.
     2. Verify that the contract processes and stores the data correctly.
   * **Expected Result**: The contract should handle and store large updates without errors.

# Testing Use Case #8: Withdraw Token Rewards

**Test**:

* + Verify that whitelisted users can withdraw their earned rewards.
  + Ensure the rewards balance is reset to zero after withdrawal.

**Implementation**:

* Implement withdrawTokenRewards to handle reward withdrawals and reset balances.

**Full unit tests:**

1. **Test Case 1: Successful Token Reward Withdrawal**
   * **Description**: Ensure that a whitelisted user can successfully withdraw their earned token rewards.
   * **Steps**:
     1. Add a user to the whitelist and allocate token rewards.
     2. Call withdrawTokenRewards for that user.
     3. Verify that the rewards are withdrawn and the balance is updated.
   * **Expected Result**: The user's rewards should be withdrawn and the accumulated balance should be reset to zero.
2. **Test Case 2: Only Whitelisted Users Can Withdraw**
   * **Description**: Ensure that only whitelisted users can withdraw their rewards.
   * **Steps**:
     1. Attempt to call withdrawTokenRewards from a non-whitelisted address.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Withdrawal of Zero Rewards**
   * **Description**: Verify that attempting to withdraw zero rewards does not affect the contract state.
   * **Steps**:
     1. Call withdrawTokenRewards with zero rewards.
     2. Verify that the contract state remains unchanged.
   * **Expected Result**: The contract state should remain unchanged.
4. **Test Case 4: Update Rewards Balance After Withdrawal**
   * **Description**: Ensure that the rewards balance is updated correctly after a withdrawal.
   * **Steps**:
     1. Add rewards to a user’s account.
     2. Withdraw rewards and check the balance update.
   * **Expected Result**: The rewards balance should be reset to zero after withdrawal.
5. **Test Case 5: Handling Large Withdrawals**
   * **Description**: Verify that large reward withdrawals are handled correctly.
   * **Steps**:
     1. Allocate a large amount of rewards to a user.
     2. Call withdrawTokenRewards and verify the transaction.
   * **Expected Result**: The large withdrawal should be processed correctly without errors.

# Testing Use Case #9: Withdraw Tokens to Burn

**Test**:

* + Verify that only the owner can withdraw tokens to burn.
  + Ensure the tokens are removed from the smart contract.

**Implementation**:

* Implement withdrawTokensToBurn to manage token withdrawals for burning.

**Full unit tests:**

1. **Test Case 1: Successful Token Burn**
   * **Description**: Ensure that the owner can successfully withdraw tokens to burn.
   * **Steps**:
     1. Call withdrawTokensToBurn with a valid amount.
     2. Verify that the tokens are withdrawn and burned.
   * **Expected Result**: Tokens should be withdrawn and burned as specified.
2. **Test Case 2: Only Owner Can Burn Tokens**
   * **Description**: Verify that only the contract owner can initiate a token burn.
   * **Steps**:
     1. Attempt to call withdrawTokensToBurn from a non-owner account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Burn More Tokens Than Available**
   * **Description**: Ensure that attempting to burn more tokens than are available fails.
   * **Steps**:
     1. Fund the contract with a specific amount of tokens.
     2. Attempt to burn more tokens than are available.
   * **Expected Result**: The transaction should revert with an error indicating insufficient tokens.
4. **Test Case 4: Verify Burned Tokens**
   * **Description**: Validate that the burned tokens are accurately recorded and removed from the contract.
   * **Steps**:
     1. Burn tokens and check the contract’s token balance.
     2. Verify that the burned tokens are no longer in the contract.
   * **Expected Result**: The token balance should reflect the burned amount.
5. **Test Case 5: Token Burn Authorization Check**
   * **Description**: Verify that only authorized tokens can be burned.
   * **Steps**:
     1. Attempt to burn tokens that are not authorized or are restricted.
   * **Expected Result**: The transaction should revert with an error indicating unauthorized tokens.

# Testing Use Case #10: Update Waiting Time

**Test**:

* + Ensure only the owner can update waiting times.
  + Verify that waiting times are updated correctly.

**Implementation**:

* Implement updateWaitingTime to handle changes in waiting periods.

**Full unit tests:**

1. **Test Case 1: Successful Update of Waiting Time**
   * **Description**: Ensure that the contract owner can successfully update the waiting time for different funding types.
   * **Steps**:
     1. Call updateWaitingTime with valid parameters.
     2. Verify that the waiting time for the specified funding type is updated correctly.
   * **Expected Result**: The waiting time should be updated as specified.
2. **Test Case 2: Only Owner Can Update Waiting Time**
   * **Description**: Verify that only the contract owner can update the waiting time.
   * **Steps**:
     1. Attempt to call updateWaitingTime from a non-owner account.
   * **Expected Result**: The transaction should revert with an authorization error.
3. **Test Case 3: Update with Invalid Parameters**
   * **Description**: Ensure that updating the waiting time with invalid parameters fails.
   * **Steps**:
     1. Call updateWaitingTime with invalid or negative values.
   * **Expected Result**: The transaction should revert with an error indicating invalid parameters.
4. **Test Case 4: Verify Waiting Time Update**
   * **Description**: Validate that the waiting time for each funding type is correctly updated.
   * **Steps**:
     1. Update waiting times for different funding types.
     2. Retrieve and verify the updated waiting times.
   * **Expected Result**: The retrieved waiting times should match the updated values.
5. **Test Case 5: Handling Large Updates**
   * **Description**: Verify the contract’s ability to handle updates to waiting times for multiple funding types.
   * **Steps**:
     1. Call updateWaitingTime for multiple funding types with different values.
     2. Verify that all updates are processed correctly.
   * **Expected Result**: All updates should be applied without errors.

# Testing External Use Case #11: Calculate Token Rewards Earned Weekly

**Test**:

* + Since this is an external script, test it separately using JavaScript/TypeScript.
  + Ensure the script correctly calculates and sends rewards to the smart contract.

**Implementation**:

* Implement the external script to calculate and send token rewards.

**Full unit tests:**

1. **Test Case 1: Successful Calculation and Update**
   * **Description**: Ensure that the script calculates and updates rewards correctly based on weekly data.
   * **Steps**:
     1. Run calculateTokenRewardsEarnedWeekly script.
     2. Verify that rewards are correctly calculated and updated in the smart contract.
   * **Expected Result**: Rewards should be accurately calculated and updated in the contract.
2. **Test Case 2: Handling Large Data Sets**
   * **Description**: Verify the script’s performance and accuracy when processing a large number of wallets and rewards.
   * **Steps**:
     1. Populate the contract with a large dataset of wallet addresses and rewards.
     2. Run calculateTokenRewardsEarnedWeekly and verify the results.
   * **Expected Result**: The script should handle large datasets efficiently and accurately.
3. **Test Case 3: Validate Weekly Calculation Accuracy**
   * **Description**: Ensure that the weekly calculations match expected results based on provided input data.
   * **Steps**:
     1. Input known reward data and expected weekly results.
     2. Run calculateTokenRewardsEarnedWeekly and compare results.
   * **Expected Result**: Calculated rewards should match the expected results.
4. **Test Case 4: Handling Edge Cases**
   * **Description**: Test the script’s behavior with edge cases such as zero rewards or very high reward values.
   * **Steps**:
     1. Input edge cases into the contract.
     2. Run calculateTokenRewardsEarnedWeekly and verify correct handling of these cases.
   * **Expected Result**: The script should handle edge cases gracefully and produce accurate results.
5. **Test Case 5: Script Execution Time**
   * **Description**: Measure the time taken by the script to perform calculations and ensure it is within acceptable limits.
   * **Steps**:
     1. Record the start time before running calculateTokenRewardsEarnedWeekly.
     2. Measure the end time after execution and calculate the duration.
   * **Expected Result**: The script should execute within an acceptable time frame, ensuring efficiency.