

Contract Management documentation

A smart contract for efficiently managing contract addresses and respective descriptions.

1. Smart Contract

a. Choice of data structure

The **contracts** mapping provides a constant-time complexity($O(1)$) for storage, updating, removal and retrieval of data using contract addresses as unique identifiers. This makes it a very efficient way to manage contract addresses and respective descriptions.

```
mapping(address => string) public Contracts;
```

The **isAuthorized** is used for access check by all external functions, making it gas efficient as possible is a priority which is resolved with the use of mapping

```
mapping(address => bool) public isAuthorized;
```

b. Choice of Access Control Mechanism:

The access control is managed with the use of an **isAuthorized** mapping and **onlyAuthorizedUser** modifier. Only authorized user have the privileges to store, update or removal contract data

Reusability: The **onlyAuthorizedUser** provides a reusable way to control access on functions, thereby reducing redundancy

Scalability: The choice of adding authorized users through a mapping allows for scalability in access control. The contract does not have a fixed or hardcoded set of authorized users, as the need arises, access privileges can be reassigned or revoked as needed without requiring changes to the contract's core functionality.

Gas Efficiency: The access control check in the **onlyAuthorizedUser** modifier is designed to have a constant time complexity, denoted as $O(1)$. That means gas cost for this check remains the same regardless of the size of the number of authorized users, or any other factors.

C. Various functions

GrantAccess: Permits contract owner to give authorized access to user addresses

- Only owner can call this function
- The **_user** address must be a valid address
- The **_user** address must be an unauthorized address

function grantAccess(address _user) external;

RevokeAccess: Permits contract owner to remove authorized access from user addresses

- Only owner can call this function
- The **_user** address must be a valid address
- The **_user** address must be an authorized address

function revokeAccess(address _user) external;

AddContract: Permits authorized users to store contract addresses with respective descriptions

- Only authorized users can call this function
- Cannot add an already existent contract address
- The **_contractAddress** address must be a valid contract address
- The description of a contract address must not be empty

function addContract(address _contractAddress, string memory _description) external;

UpdateContractDescription: Permits authorized users to update the description of a stored contract address

- Only authorized users can call this function
- The **_contractAddress** address must be a valid address
- Cannot update a contract that has not been stored
- The description of a contract address must not be empty

function updateContractDescription(address _contractAddress, string memory _newDescription) external;

RemoveContract: Permits authorized users to remove stored contract addresses with respective description.

- Only authorized users can call this function

- The **_contractAddress** address must be a valid address
- Cannot remove a contract that has not been stored

function removeContract(address _contractAddress) external;

IsValidContractAddress: is an internal function that permits to check if an address is a valid contract address and returns a true/false result

function _isValidContractAddress(address _contractAddress) internal;

D. Modifiers

isOwner: permits to restrict access to the main contract owner only.

modifier isOwner(){}

isAuthorizedUser: permits to restrict access to authorized users only

modifier isAuthorizedUser(){}

isValidAddress: permits to check for address validity

modifier isValidAddress(address _contractAddress){}

E. Events

Events serve as a notifier to the external world about specific occurrences or state changes within the contract.

- Event sent when new contract addresses are stores or updated

event ContractInfo(address dataOperator, address contractAddress, string description,string status);

- Event sent when stored contract addresses are removed

event ContractRemoved(address contractAddress);

- Event sent when user access are granted or revoked

event AccessInfo(address user, string description);

2. Test Approach

The main objective of testing the **ContractManager** is to ensure that the contract's functions work as expected, adhere to access control mechanisms, and handle edge cases.

Test Cases

a. Access control management

Grant Access:

- Operation should revert if **ContractManager** owner is not function caller
- Operation should revert if provided user address is invalid
- Operation should revert if user address provided already has authorized access
- Operation should successfully grant access to a user if all conditions are respected

Revoke Access:

- Operation should revert if caller is not contract owner
- Operation should revert if user address provided is invalid
- Operation should revert if user address provided doesn't have authorized access
- Operation should successfully revoke access for a user if all conditions are met

b. Contract storage management

Add Contract:

- Operation should revert if caller is unauthorized
- Operation should revert if provided contract address is an invalid address
- Operation should revert if trying to store a non existent contract address
- Operation should revert if trying to add a contract address with empty description
- Operation should revert if trying to save a contract address that already exist
- Operation should successfully store a contract address with respective description if all conditions are respected

Update Contract description:

- Operation should revert if caller is unauthorized
- Operation should revert if provided contract address is an invalid address
- Operation should revert if trying to update a non existent contract
- Operation should revert if trying to update a contract with empty description
- Operation should successfully update a contract description if all conditions are respected

Remove stored Contract:

- Operation should revert if caller is unauthorized
- Operation should revert if provided contract address is an invalid address
- Operation should revert if trying to remove a non existent contract
- Operation should successfully remove a contract

Test Environment:

- Use Hardhat as the testing framework.
- Ensure the local blockchain environment is set up for contract deployment and testing.

Test Execution:

- Automated Testing:**
 - Used automated testing script to execute test cases.
 - Ensure that automated tests cover all edge cases and scenarios.
- Manual Testing:**
 - Conduct manual testing for edge cases and complex scenarios.
 - Manually checked the gas costs associated with critical functions.
- Code Review:**
 - Perform a code review of the test scripts to ensure accuracy and coverage.
 - Verify that the test scripts align with the documented test approach.
- Gas Profiling:**
 - Use Hardhat's gas profiler to analyze gas consumption during test executions.
 - Made sure that the gas costs remain within acceptable range.