Registering Accounts

In the previous tutorial, you looked at how to mint an initial circulating supply of tokens and how you could log events as per theevents standard . You then deployed the contract and saw the FTs in your wallet balance. In this tutorial, you'll learn about thestorage management standard and how you can register accounts in your FT contract in order to prevent a malicious party from draining your contract of all its funds.

Introduction

Whenever a new person receives any fungible tokens, they're added to theaccounts lookup map on the contract. By doing this, you're adding bytes to the contract. If you made it so any user could receive FTs for free, that system could easily be abused. Users could essentially "drain" the contract of all it's funds by sending small amounts of FTs to many accounts. For this reason, you'll want to charge users for the information they're storing and the bytes they're using on the contract. This way of charging users, however, should be standardized so it works across all contracts.

Enter thestorage management standard

Storage Management Standard

The storage management standard is a set of rules that govern how a contract should charge users for storage. It outlines functions and behaviors such that all contracts implementing the standard are interoperable with each other. The 3 functions you'll need to implement are:

- storage_deposit
-
    - Allows a user to deposit some amount of storage to the contract. If the user over deposits, they're refunded for the excess NEAR.
- storage_balance_of
-
    - Query for the storage paid by a given user
- storage_balance_bounds
-
    - Query for the minimum and maximum amount of storage needed to interact with a given contract.

With these functions outlined, you could make a reasonable assumption that the flow would be:

1. If a user doesn't exist on the contract, they need to deposit some storage to cover the bytes they use.
2. Once the user deposits NEAR via thestorage_deposit
3. function, they're free to interact with the contract.

You might be asking yourself what the deposit amount should be. There are two ways you can go about getting this information:

- Dynamically calculate the bytes every individual user would take up in thestorage_deposit
- function by inserting them intoaccounts
- map, measuring the bytes, and then removing them from the map after.
- Calculate the bytes for inserting a largest possible account ID once upon initializing the contract and simply charge every user the same amount.

For the purpose of simplicity, we'll assume the second method.

Modifications to the Contract

This "bytes for longest account ID" should be stored in the contract's state such that we can pull the value during thestorage_deposit function and ensure the user attaches enough NEAR. Open thesrc/lib.rs file and add the following code to theContract struct. If you're just joining us now, you can find the skeleton code for this tutorial in the3.initial-supply folder.

4.storage/src/lib.rs loading ... See full example on GitHub You'll now need a way to calculate this amount which will be done in the initialization function. Move to thesrc/internal.rs file and add the following private functionmeasure_bytes_for_longest_account_id which will add the longest possible account ID and remove it while measuring how many bytes the operation took. It will then set thebytes_for_longest_account_id field to the result.

4.storage/src/internal.rs loading ... See full example on GitHub You'll also want to create a function for "registering" an account after they've paid for storage. To do this, you can simply insert them into theaccounts map with a balance of 0. This way, you know that any account currently in the map is considered "registered" and have paid for storage. Any account that attempts to receive FTs must be in the map with a balance of 0 or greater. If they aren't, the contract should throw.

4.storage/src/internal.rs loading ... See full example on GitHub Let's also create a function to panic with a custom message if

the user doesn't exist yet.

4.storage/src/internal.rs loading ... See full example on GitHub Now when you call theinternal_deposit function, rather than defaulting the user's balance to0 if they don't exist yet via:

let balance =

self . accounts . get (& account_id) . unwrap_or (0) ; You can replace it with the following:

4.storage/src/internal.rs loading ... See full example on GitHub With this finished, yourinternal.rs should look as follows:

use

near_sdk :: { require } ;

use

crate :: * ;

impl

Contract

{ pub (crate)

fn

internal_unwrap_balance_of (& self , account_id :

& AccountId)

->

Balance

{ ... }

pub (crate)

fn

internal_deposit (& mut

self , account_id :

& AccountId , amount :

Balance)

{ ... }

pub (crate)

fn

internal_register_account (& mut

self , account_id :

& AccountId)

{ ... }

pub (crate)

fn

measure_bytes_for_longest_account_id (& mut

self)

{ ... } } There's only one problem you need to solve with this. When initializing the contract, the implementation will throw. This is because you callinternal_deposit and the owner doesn't have a balance yet. To fix this, let's modify the initialization function to register the owner before depositing the FTs in their account. In addition, you should measure the bytes for the

registration in this function.

4.storage/src/lib.rs loading ... See full example on GitHub

Implementing Storage Standard

With this finished, you're now ready to implement the storage management standard. If you remember, the three functions you'll be implementing, we can break each down into their core functionality and decide how to proceed.

- storage_balance_bounds
-
    - Query for the minimum and maximum amount of storage needed to interact with a given contract.

Since you're creating a fungible token contract and the storage price won't change (unless the NEAR cost per byte changes), the minimum and maximum storage costs should be the same. These values should be equal to the amount of bytes for the longest account ID you calculated in themeasure_bytes_for_longest_account_id function multiplied by the current NEAR price per byte. Switch to thesrc/storage.rs file to get started.

4.storage/src/storage.rs loading ... See full example on GitHub* storage_balance_of * - Query for the storage paid by a given user.

As we mentioned earlier, you can tell if somebody has paid for storage by checking if they're in theaccounts map. If they are, you know that they've paid the amount returned bystorage_balance_bounds .

4.storage/src/storage.rs loading ... See full example on GitHub* storage_deposit * - Allows a user to deposit some amount of storage to the contract. If the user over deposits, they're refunded for the excess NEAR.

In order to implement this logic, you first need to get the attached deposit. You'll also need to ensure that the user hasn't already paid for storage (i.e. they're in theaccounts map). If they are, simply refund the caller for the NEAR they attached to the call.

If the user isn't registered yet, you should get the storage cost by callingstorage_balance_bounds and make sure they've attached enough to cover that amount. Once this if finished, you can register them and refund any excess NEAR.

4.storage/src/storage.rs loading ... See full example on GitHubWith this finished, you're ready to build and deploy the contract!

Deploying the contract

Since the current contract you have is already initialized, let's create a sub-account and deploy to again.

Creating a sub-account

Run the following command to create a sub-accountstorage of your main account with an initial balance of 25 NEAR which will be transferred from the original to your new account.

near create-account storage.FT_CONTRACT_ID --masterAccount FT_CONTRACT_ID --initialBalance 25 Next, you'll want to export an environment variable for ease of development:

export STORAGE_FT_CONTRACT_ID=storage.FT_CONTRACT_ID Using the build script, build the deploy the contract as you did in the previous tutorials:

cd 3.initial-supply && ./build.sh && cd .. && near deploy STORAGE_FT_CONTRACT_ID out/contract.wasm

Initialization

Now that the contract is deployed, it's time to initialize it and mint the total supply. Let's once again create an initial supply of 1000gtNEAR .

near call STORAGE_FT_CONTRACT_ID new_default_meta '{"owner_id": "'STORAGE_FT_CONTRACT_ID'", "total_supply": "1000000000000000000000000000"}' --accountId STORAGE_FT_CONTRACT_ID If you now query for the storage paid by the owner, you should see that they're registered!

near view STORAGE_FT_CONTRACT_ID storage_balance_of '{"account_id": "'STORAGE_FT_CONTRACT_ID'"}' This should return a struct. The Total amount is roughly0.00125 NEAR to register and the user has 0 available NEAR since it's all being used up to pay for registration.

{ total: '1250000000000000000000', available: '0' } You can also query for the storage balance required to interact with the contract:

near view STORAGE_FT_CONTRACT_ID storage_balance_bounds You'll see that it returns the same amount as thestorage_balance_of query above with the min equal to the max:

{ min: '1250000000000000000000', max: '1250000000000000000000' }

Conclusion

Today you went through and created the logic for registering users on the contract. This logic adheres to thestorage management standard and is customized to meet our needs when writing a FT contract. You then built, deployed, and tested those changes. In thenext tutorial , you'll look at the basics of how to transfer FTs to other usersEdit this page Last updatedonMar 9, 2024 byFrank Was this page helpful? Yes No

Previous Circulating Supply Next Transferring FTs