

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 the [events standard](#). You then deployed the contract and saw the FTs in your wallet balance. In this tutorial, you'll learn about the [storage 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 the accounts 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 its 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 the [storage 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 the `storage_deposit` 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 the `storage_deposit` function by inserting them into accounts
- 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 the `storage_deposit` function and ensure the user attaches enough NEAR. Open the `src/lib.rs` file and add the following code to the `Contract` struct. If you're just joining us now, you can find the skeleton code for this tutorial in the `3.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 the `src/internal.rs` file and add the following private function `measure_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 the `bytes_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 the accounts 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 the `internal_deposit` function, rather than defaulting the user's balance to 0 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, your `internal.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 call `internal_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

4.storage/src/lib.rs loading ... [See full example on GitHub](#)

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

{ total: '12500000000000000000', 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 the storage_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 the [storage management standard](#) and is customized to meet our needs when writing a FT contract. You then built, deployed, and tested those changes. In the [next tutorial](#), you'll look at the basics of how to transfer FTs to other users [Edit this page](#) Last updated on Mar 9, 2024 by Frank Was this page helpful? Yes No

[Previous Circulating Supply](#) [Next Transferring FTs](#)