Defining a Fungible Token

This is the first of many tutorials in a series where you'll be creating a complete FT smart contract from scratch that conforms with all the NEARFT standards . Today you'll learn what a Fungible Token is and how you can define one on the NEAR blockchain. You will be modifying a bare-boneskeleton smart contract by filling in the necessary code snippets needed to add this functionality.

Introduction

To get started, switch to the1.skeleton folder in our repo. If you haven't cloned the repository, refer to theContract Architecture to get started.

If you wish to see the finished code for this portion of the tutorial, that can be found on the2.defining-a-token folder.

Modifications to the skeleton contract

At its very core, a fungible token is an exchangeable asset thatis divisible but isnot unique . For example, if Benji had 1 Canadian dollar, it would be worth the exact same as Matt's Canadian dollar. Both their dollars are fungible and exchangeable. In this case, the fungible token is the canadian dollar. All fiat currencies are fungible and exchangeable.

Non-fungible tokens, on the other hand, areunique andindivisible such as a house or a car. Youcannot have another asset that is exactly the same. Even if you had a specific car model, such as a Corvette 1963 C2 Stingray, each car would have a separate serial number with a different number of kilometers driven etc...

Now that you understand what a fungible token is, let's look at how you can define one in the contract itself.

Defining a fungible token

Start by navigating to the1.skeleton/src/metadata.rs file. This is where you'll define the metadata for the fungible token itself. There are several ways NEAR allows you to customize your token, all of which are found in themetadata standard. Let's break them up into the optional and non-optional portions.

Required:

- spec
- : Indicates the version of the standard the contract is using. This should be set toft-1.0.0
- .
- name
- : The human readable name of the token such as "Wrapped NEAR" or "TEAM Tokens".
- symbol
- : The abbreviation of the token such aswNEAR
- orgtNEAR
- .
- decimals
- : used in frontends to show the proper significant digits of a token. This concept is explained well in thisOpenZeppelin post
- .

Optional:

- icon
- : The image for the token (must be adata URL
-).
- reference
- : A link to any supplementary JSON details for the token stored off-chain.
- reference_hash
- : A hash of the referenced JSON.

With this finished, you can now add these fields to the metadata in the contract.

2.define-a-token/src/metadata.rs loading ... See full example on GitHub Now that you've defined what the metadata will look like, you need someway to store it on the contract. Switch to the1.skeleton/src/lib.rs file and add the following to theContract struct. You'll want to store the metadata on the contract under themetadata field.

2.define-a-token/src/lib.rs loading ... See full example on GitHub You've now definedwhere the metadata will live but you'll also need someway to pass in the metadata itself. This is where the initialization function comes into play.

Initialization Functions

You'll now create what's called an initialization function; you can name itnew . This function needs to be invoked when you first deploy the contract. It will initialize all the contract's fields that you've defined with default values. It's important to note that youcannot call these methods more than once.

2.define-a-token/src/lib.rs loading ... See full example on GitHubMore often than not when doing development, you'll need to deploy contracts several times. You can imagine that it might get tedious to have to pass in metadata every single time you want to initialize the contract. For this reason, let's create a function that can initialize the contract with a set of defaultmetadata . You can call itnew_default_meta .

2.define-a-token/src/lib.rs loading ... See full example on GitHubThis function is simply calling the previousnew function and passing in some default metadata behind the scenes.

At this point, you've defined the metadata for your fungible tokens and you've created a way to store this information on the contract. The last step is to introduce a getter that will query for and return the metadata. Switch to the1.skeleton/src/metadata.rs file and add the following code to theft_metadata function.

2.define-a-token/src/metadata.rs loading ... See full example on GitHubThis function will get themetadata object from the contract which is of typeFungibleTokenMetadata and will return it.

Interacting with the contract on-chain

Now that the logic for defining a custom fungible token is complete and you've added a way to query for the metadata, it's time to build and deploy your contract to the blockchain.

Deploying the contract

We've included a very simple way to build the smart contracts throughout this tutorial using a bash script. The following command will build the contract and copy over the.wasm file to a folderout/contract.wasm . The build script can be found in the1.skeleton/build.sh file.

cd 1.skeleton && ./build.sh && cd .. There will be a list of warnings on your console, but as the tutorial progresses, these warnings will go away. You should now see the folderout/ with the filecontract.wasm inside. This is what we will be deploying to the blockchain.

For deployment, you will need a NEAR account with the keys stored on your local machine. Navigate to theNEAR wallet site and create an account.

info Please ensure that you deploy the contract to an account with no pre-existing contracts. It's easiest to simply create a new account or create a sub-account for this tutorial. Log in to your newly created account withnear-cli by running the following command in your terminal.

near login To make this tutorial easier to copy/paste, we're going to set an environment variable for your account ID. In the command below, replaceYOUR_ACCOUNT_NAME with the account name you just logged in with including the.testnet portion:

export FT_CONTRACT_ID="YOUR_ACCOUNT_NAME" Test that the environment variable is set correctly by running:

echo FT_CONTRACT_ID Verify that the correct account ID is printed in the terminal. If everything looks correct you can now deploy your contract. In the root of your FT project run the following command to deploy your smart contract.

near deploy FT_CONTRACT_ID out/contract.wasm At this point, the contract should have been deployed to your account and you're ready to move onto creating your personalized fungible token.

Creating the fungible token

The very first thing you need to do once the contract has been deployed is to initialize it. For simplicity, let's call the default metadata initialization function you wrote earlier so that you don't have to type the metadata manually in the CLI.

near call FT_CONTRACT_ID new_default_meta '{"owner_id": "'FT_CONTRACT_ID'", "total_supply": "0"}' --accountId FT_CONTRACT_ID

Viewing the contract's metadata

Now that the contract has been initialized, you can query for the metadata by calling the function you wrote earlier.

near view FT_CONTRACT_ID ft_metadata This should return an output similar to the following:

{ spec: 'ft-1.0.0', name: 'Team Token FT Tutorial', symbol: 'gtNEAR', icon: ' / *...lots of base64 data...*/ j4Mvhy9H9NlnieJ4iwoo9ZlyLGx4pnrPWeB4CVGRZZcJ7Vohwhi0z5MJY4cVL4MdP/Z', reference: null, reference_hash: null,

decimals: 24 } Go team! You've now verified that everything works correctly and you've defined your own fungible token!

In the next tutorial, you'll learn about how to create a total supply and view the tokens in the wallet.

Conclusion

In this tutorial, you went through the basics of setting up and understanding the logic behind creating a fungible token on the blockchain using a skeleton contract.

You first looked atwhat a fungible token is and how it differs from a non-fungible token. You then learned how to customize and create your own fungible tokens and how you could modify the skeleton contract to achieve this. Finally you built and deployed the contract and interacted with it using the NEAR CLI.

Next Steps

In thenext tutorial , you'll find out how to create an initial supply of tokens and have them show up in the NEAR walletEdit this page Last updatedonMar 8, 2024 byFrank Was this page helpful? Yes No

Previous Contract Architecture Next Circulating Supply