Royalty

In this tutorial you'll continue building your non-fungible token (NFT) smart contract, and learn how to implement perpetual royalties into your NFTs. This will allow people to get a percentage of the purchase price when an NFT is sold.

Introduction

By now, you should have a fully fledged NFT contract, except for the royalties support. To get started, either switch to the5.approval branch from ourGitHub repository , or continue your work from the previous tutorials.

git checkout 5.approval tip If you wish to see the finished code for thisRoyalty tutorial, you can find it on the6.royalty branch.

Thinking about the problem

In order to implement the functionality, you first need to understand how NFTs are sold. In the previous tutorial, you saw how someone with an NFT could list it on a marketplace using thenft_approve function by passing in a message that could be properly decoded. When a user purchases your NFT on the marketplace, what happens?

Using the knowledge you have now, a reasonable conclusion would be to say that the marketplace transfers the NFT to the buyer by performing a cross-contract call and invokes the NFT contract'snft_transfer method. Once that function finishes, the marketplace would pay the seller for the correct amount that the buyer paid.

Let's now think about how this can be expanded to allow for a cut of the pay going to other accounts that aren't just the seller.

Expanding the current solution

Since perpetual royalties will be on a per-token basis, it's safe to assume that you should be changing theToken andJsonToken structs. You need some way of keeping track of what percentage each account with a royalty should have. If you introduce a map of an account to an integer, that should do the trick.

Now, you need some way to relay that information to the marketplace. This method should be able to transfer the NFT exactly like the old solution but with the added benefit of telling the marketplace exactly what accounts should be paid what amounts. If you implement a method that transfers the NFT and then calculates exactly what accounts get paid and to what amount based on a passed-in balance, that should work nicely.

This is what theroyalty standards outlined. Let's now move on and modify our smart contract to introduce this behavior.

Modifications to the contract

The first thing you'll want to do is add the royalty information to the structs. Open thenft-contract/src/metadata.rs file and addroyalty to theToken struct:

pub royalty :

HashMap < AccountId ,

u32

, Second, you'll want to addroyalty to theJsonToken struct as well:

pub royalty :

HashMap < AccountId ,

u32

,

Internal helper function

royalty_to_payout

To simplify the payout calculation, let's add a helperroyalty_to_payout function tosrc/internal.rs . This will convert a percentage to the actual amount that should be paid. In order to allow for percentages less than 1%, you can give 100% a value of10,000 . This means that the minimum percentage you can give out is 0.01%, or1 . For example, if you wanted the accountbenji.testnet to have a perpetual royalty of 20%, you would insert the pair"benji.testnet": 2000 into the payout map.

nft-contract/src/internal.rs loading ... See full example on GitHub If you were to use theroyalty_to_payout function and pass in2000 as theroyalty_percentage and anamount_to_pay of 1 NEAR, it would return a value of 0.2 NEAR.

Royalties

nft_payout

Let's now implement a method to check what accounts will be paid out for an NFT given an amount, or balance. Open thenft-contract/src/royalty.rs file, and modify thenft_payout function as shown.

nft-contract/src/royalty.rs loading ... See full example on GitHub This function will loop through the token's royalty map and take the balance and convert that to a payout using theroyalty_to_payout function you created earlier. It will give the owner of the token whatever is left from the total royalties. As an example:

You have a token with the following royalty field:

Token

{ owner_id :

"damian" , royalty :

{ "benji" :

1000 , "josh" :

500 , "mike" :

2000 } } If a user were to callnft_payout on the token and pass in a balance of 1 NEAR, it would loop through the token's royalty field and insert the following into the payout object:

Payout

{ payout :

{ "benji" :

0.1

NEAR , "josh" :

0.05

NEAR , "mike" :

0.2

NEAR } } At the very end, it will insertdamian into the payout object and give him1 NEAR - 0.1 - 0.05 - 0.2 = 0.65 NEAR .

nft_transfer_payout

Now that you know how payouts are calculated, it's time to create the function that will transfer the NFT and return the payout to the marketplace.

nft-contract/src/royalty.rs loading ... See full example on GitHub

Perpetual royalties

To add support for perpetual royalties, let's edit thesrc/mint.rs file. First, add an optional parameter for perpetual royalties. This is what will determine what percentage goes to which accounts when the NFT is purchased. You will also need to create and insert the royalty to be put in theToken object:

nft-contract/src/mint.rs loading ... See full example on GitHub Next, you can use the CLI to query the newnft_payout function and validate that it works correctly.

Adding royalty object to struct implementations

Since you've added a new field to yourToken andJsonToken structs, you need to edit your implementations accordingly. Move to thenft-contract/src/internal.rs file and edit the part of yourinternal_transfer function that creates the newToken object:

nft-contract/src/internal.rs loading ... See full example on GitHub Once that's finished, move to thenft-contract/src/nft_core.rs

file. You need to edit your implementation ofnft_token so that theJsonToken sends back the new royalty information.

nft-contract/src/nft_core.rs loading ... See full example on GitHub

Deploying the contract

As you saw in the previous tutorial, adding changes like these will cause problems when redeploying. Since these changes affect all the other tokens and the state won't be able to automatically be inherited by the new code, simply redeploying the contract will lead to errors. For this reason, you'll create a new account again.

Deployment

Next, you'll deploy this contract to the network.

export ROYALTY_NFT_CONTRACT_ID= near create-account ROYALTY_NFT_CONTRACT_ID --useFaucet Using the build script, build the deploy the contract as you did in the previous tutorials:

yarn build && near deploy ROYALTY_NFT_CONTRACT_ID out/main.wasm

Initialization and minting

Since this is a new contract, you'll need to initialize and mint a token. Use the following command to initialize the contract:

near call ROYALTY_NFT_CONTRACT_ID new_default_meta '{"owner_id": "'ROYALTY_NFT_CONTRACT_ID'"}' --accountId ROYALTY_NFT_CONTRACT_ID Next, you'll need to mint a token. By running this command, you'll mint a token with a token ID"royalty-token" and the receiver will be your new account. In addition, you're passing in a map with two accounts that will get perpetual royalties whenever your token is sold.

near call ROYALTY_NFT_CONTRACT_ID nft_mint '{"token_id": "royalty-token", "metadata": {"title": "Royalty Token", "description": "testing out the new royalty extension of the standard", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}, "receiver_id": "'ROYALTY_NFT_CONTRACT_ID'", "perpetual_royalties": {"benjiman.testnet": 2000, "mike.testnet": 1000, "josh.testnet": 500}}' --accountId ROYALTY_NFT_CONTRACT_ID --amount 0.1 You can check to see if everything went through properly by calling one of the enumeration functions:

near view ROYALTY_NFT_CONTRACT_ID nft_tokens_for_owner '{"account_id": "'ROYALTY_NFT_CONTRACT_ID'", "limit": 10}' This should return an output similar to the following:

[{ "token_id": "royalty-token", "owner_id": "royalty.goteam.examples.testnet", "metadata": { "title": "Royalty Token", "description": "testing out the new royalty extension of the standard", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif", "media_hash": null, "copies": null, "issued_at": null, "expires_at": null, "starts_at": null, "updated_at": null, "extra": null, "reference": null, "reference_hash": null }, "approved_account_ids": {}, "royalty": { "josh.testnet": 500, "benjiman.testnet": 2000, "mike.testnet": 1000 } }] Notice how there's now a royalty field that contains the 3 accounts that will get a combined 35% of all sales of this NFT? Looks like it works! Go team :)

NFT payout

Let's calculate the payout for the"royalty-token" NFT, given a balance of 100 yoctoNEAR. It's important to note that the balance being passed into thenft_payout function is expected to be in yoctoNEAR.

near view ROYALTY_NFT_CONTRACT_ID nft_payout '{"token_id": "royalty-token", "balance": "100", "max_len_payout": 100}' This command should return an output similar to the following:

{ payout: { 'josh.testnet': '5', 'royalty.goteam.examples.testnet': '65', 'mike.testnet': '10', 'benjiman.testnet': '20' } } If the NFT was sold for 100 yoctoNEAR, josh would get 5, benji would get 20, mike would get 10, and the owner, in this caseroyalty.goteam.examples.testnet would get the rest: 65.

Conclusion

At this point you have everything you need for a fully functioning NFT contract to interact with marketplaces. The last remaining standard that you could implement is the events standard. This allows indexers to know what functions are being called and makes it easier and more reliable to keep track of information that can be used to populate the collectibles tab in the wallet for example.

remember If you want to see the finished code from this tutorial, you can checkout the6.royalty branch. Versioning for this article At the time of this writing, this example works with the following versions:

- near-cli:4.0.4

- NFT standard:NEP171
- , version1.1.0
- Enumeration standard:NEP181
- , version1.0.0
- Royalties standard:NEP199
- , version2.0.0 Edit this page Last updatedonFeb 16, 2024 bygarikbesson Was this page helpful? Yes No

Previous Approvals Next Events