Marketplace

In this tutorial, you'll learn the basics of an NFT marketplace contract where you can buy and sell non-fungible tokens for NEAR. In the previous tutorials, you went through and created a fully fledged NFT contract that incorporates all the standards found in theNFT standard .

Introduction

Throughout this tutorial, you'll learn how a marketplace contract could work on NEAR. This is meant to be an example and there is no canonical implementation. Feel free to branch off and modify this contract to meet your specific needs.

Using the same repository as the previous tutorials, if you checkout the8.marketplace branch, you should have the necessary files to complete the tutorial.

git checkout 8.marketplace

File structure

The changes made include a new root level directory calledmarket-contract . This contains both the build script, dependencies as well as the actual contract code as outlined below.

market-contract ├─── Cargo.lock ├─── Cargo.toml ├─── README.md ├─── build.sh └─── src ├─── external.rs ├─── internal.rs ├─── lib.rs ├─── nft_callbacks.rs ├─── sale.rs └─── sale_views.rs Usually, when doing work on multiple smart contracts that all pertain to the same repository, it's a good idea to structure them in their own folders as done in this tutorial. To make your work easier when building the smart contracts, we've also modified the repository'spackage.json file so that building both smart contracts can be easily done by running the following command.

yarn build This will install the dependencies for both contracts and compile them towasm files that are stored in the following directory.

nft-tutorial └─── out ├─── main.wasm └─── market.wasm

Understanding the contract

At first, the contract can be quite overwhelming but if you strip away all the fluff and dig into the core functionalities, it's actually quite simple. This contract was designed for only one thing - to allow people to buy and sell NFTs for NEAR. This includes the support for paying royalties, updating the price of your sales, removing sales and paying for storage.

Let's go through the files and take note of some of the important functions and what they do.

lib.rs

This file outlines what information is stored on the contract as well as some other crucial functions that you'll learn about below.

Initialization function

The first function you'll look at is the initialization function. This takes anowner_id as the only parameter and will default all the storage collections to their default values.

market-contract/src/lib.rs loading ... See full example on GitHub

Storage management model

Next, let's talk about the storage management model chosen for this contract. On the NFT contract, users attached NEAR to the calls that needed storage paid for. For example, if someone was minting an NFT, they would need to attachx amount of NEAR to cover the cost of storing the data on the contract.

On this marketplace contract, however, the storage model is a bit different. Users will need to deposit NEAR onto the marketplace to cover the storage costs. Whenever someone puts an NFT for sale, the marketplace needs to store that information which costs NEAR. Users can either deposit as much NEAR as they want so that they never have to worry about storage again or they can deposit the minimum amount to cover 1 sale on an as-needed basis.

You might be thinking about the scenario when a sale is purchased. What happens to the storage that is now being released on the contract? This is why we've introduced a storage withdrawal function. This allows users to withdraw any excess storage that is not being used. Let's go through some scenarios to understand the logic. The required storage for 1

sale is 0.01 NEAR on the marketplace contract.

Scenario A

- Benji wants to list his NFT on the marketplace but has never paid for storage.
- He deposits exactly 0.01 NEAR using thestorage_deposit
- method. This will cover 1 sale.
- He lists his NFT on the marketplace and is now using up 1 out of his prepaid 1 sales and has no more storage left. If he were to callstorage_withdraw
- , nothing would happen.
- Dorian loves his NFT and quickly purchases it before anybody else can. This means that Benji's sale has now been taken down (since it was purchased) and Benji is using up 0 out of his prepaid 1 sales. In other words, he has an excess of 1 sale or 0.01 NEAR.
- Benji can now callstorage_withdraw
- and will be transferred his 0.01 NEAR back. On the contract's side, after withdrawing, he will have 0 sales paid for and will need to deposit storage before trying to list anymore NFTs.

Scenario B

- Dorian owns one hundred beautiful NFTs and knows that he wants to list all of them.
- To avoid having to callstorage_deposit
- everytime he wants to list an NFT, he calls it once. Since Dorian is a baller, he attaches 10 NEAR which is enough to cover 1000 sales. Then he lists his 100 NFTs and now he has an excess of 9 NEAR or 900 sales.
- Dorian needs the 9 NEAR for something else but doesn't want to take down his 100 listings. Since he has an excess of 9 NEAR, he can easily withdraw and still have his 100 listings. After callingstorage_withdraw
- and being transferred 9 NEAR, he will have an excess of 0 sales.

With this behavior in mind, the following two functions outline the logic.

market-contract/src/lib.rs loading ... See full example on GitHub In this contract, the storage required for each sale is 0.01 NEAR but you can query that information using thestorage_minimum_balance function. In addition, if you wanted to check how much storage a given account has paid, you can query thestorage_balance_of function.

With that out of the way, it's time to move onto thenft_callbacks.rs file where you'll look at how NFTs are put for sale.

nft_callbacks.rs

This file is responsible for the logic used to put NFTs for sale. If you remember from themarketplaces section of the approvals tutorial, when users callnft_approve and pass in a message, it will perform a cross-contract call to thereceiver_id 's contract and call the methodnft_on_approve . Thisnft_callbacks.rs file will implement that function.

Listing logic

The first important thing to note is theSaleArgs struct. This is what the market contract is expecting the message that the user passes intonft_approve on the NFT contract to be. This outlines the sale price in yoctoNEAR for the NFT that is listed.

market-contract/src/nft_callbacks.rs loading ... See full example on GitHub Next, we'll look at thenft_on_approve function which is called via a cross-contract call by the NFT contract. This will make sure that the signer has enough storage to cover adding another sale. It will then attempt to get theSaleArgs from the message and create the listing.

market-contract/src/nft_callbacks.rs loading ... See full example on GitHub

sale.rs

Now that you're familiar with the process of both adding storage and listing NFTs on the marketplace, let's go through what you can do once a sale has been listed. Thesale.rs file outlines the functions for updating the price, removing, and purchasing NFTs.

Sale object

It's important to understand what information the contract is storing for each sale object. Since the marketplace has many NFTs listed that come from different NFT contracts, simply storing the token ID would not be enough to distinguish between different NFTs. This is why you need to keep track of both the token ID and the contract by which the NFT came from. In addition, for each listing, the contract must keep track of the approval ID it was given to transfer the NFT. Finally, the owner and sale conditions are needed.

market-contract/src/sale.rs loading ... See full example on GitHub

Removing sales

In order to remove a listing, the owner must call theremove_sale function and pass the NFT contract and token ID. Behind the scenes, this calls theinternal_remove_sale function which you can find in theinternal.rs file. This will assert one yoctoNEAR for security reasons.

market-contract/src/sale.rs loading ... See full example on GitHub

Updating price

In order to update the list price of a token, the owner must call theupdate_price function and pass in the contract, token ID, and desired price. This will get the sale object, change the sale conditions, and insert it back. For security reasons, this function will assert one yoctoNEAR.

market-contract/src/sale.rs loading ... See full example on GitHub

Purchasing NFTs

For purchasing NFTs, you must call theoffer function. It takes annft_contract_id andtoken_id as parameters. You must attach the correct amount of NEAR to the call in order to purchase. Behind the scenes, this will make sure your deposit is greater than the list price and call a private methodprocess_purchase which will perform a cross-contract call to the NFT contract to invoke thenft_transfer_payout function. This will transfer the NFT using theapproval management standard that you learned about and it will return thePayout object which includes royalties.

The marketplace will then callresolve_purchase where it will check for malicious payout objects and then if everything went well, it will pay the correct accounts.

market-contract/src/sale.rs loading ... See full example on GitHub

sale_view.rs

The final file we'll go through is thesale_view.rs file. This is where some of the enumeration methods are outlined. It allows users to query for important information regarding sales.

Deployment

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

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

near deploy MARKETPLACE_CONTRACT_ID out/market.wasm

Initialization and minting

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

near call MARKETPLACE_CONTRACT_ID new '{"owner_id": "'MARKETPLACE_CONTRACT_ID'"}' --accountId MARKETPLACE_CONTRACT_ID

Total supply

To query for the total supply of NFTs listed on the marketplace, you can call theget_supply_sales function. An example can be seen below.

near view MARKETPLACE_CONTRACT_ID get_supply_sales

Total supply by owner

To query for the total supply of NFTs listed by a specific owner on the marketplace, you can call theget_supply_by_owner_id function. An example can be seen below.

near view MARKETPLACE_CONTRACT_ID get_supply_by_owner_id '{"account_id": "benji.testnet"}'

Total supply by contract

To query for the total supply of NFTs that belong to a specific contract, you can call theget_supply_by_nft_contract_id function. An example can be seen below.

near view MARKETPLACE_CONTRACT_ID get_supply_by_nft_contract_id '{"nft_contract_id": "fayyr-nft.testnet"}'

Query for listing information

To query for important information for a specific listing, you can call theget_sale function. This requires that you pass in thenft_contract_token . This is essentially the unique identifier for sales on the market contract as explained earlier. It consists of the NFT contract followed by aDELIMITER followed by the token ID. In this contract, theDELIMITER is simply a period:. . An example of this query can be seen below.

near view MARKETPLACE_CONTRACT_ID get_sale '{"nft_contract_token": "fayyr-nft.testnet.token-42"}' In addition, you can query for paginated information about the listings for a given owner by calling theget_sales_by_owner_id function.

near view MARKETPLACE_CONTRACT_ID get_sales_by_owner_id '{"account_id": "benji.testnet", "from_index": "5", "limit": 10}' Finally, you can query for paginated information about the listings that originate from a given NFT contract by calling theget_sales_by_nft_contract_id function.

near view MARKETPLACE_CONTRACT_ID get_sales_by_nft_contract_id '{"nft_contract_id": "fayyr-nft.testnet", "from_index": "5", "limit": 10}'

Conclusion

In this tutorial, you learned about the basics of a marketplace contract and how it works. You went through thelib.rs file and learned about theinitialization function in addition to thestorage management model.

You then went through thenft_callbacks file to understand how todist NFTs . In addition, you went through some important functions needed for after you've listed an NFT. This includesremoving sales ,updating the price , andpurchasing NFTs .

Finally, you went through the enumeration methods found in thesale_view file. These allow you to query for important information found on the marketplace contract.

You should now have a solid understanding of NFTs and marketplaces on NEAR. Feel free to branch off and expand on these contracts to create whatever cool applications you'd like. In thenext tutorial , you'll learn how to take the existing NFT contract and optimize it to allow for:

- Lazy Minting
- Creating Collections
- Allowlisting functionalities
- Optimized Storage Models

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 Edit this page Last updatedonFeb 16, 2024 bygarikbesson Was this page helpful? Yes No

Previous Events Next Lazy Minting, Collections, and More!