Customizing the NFT Contract

In this tutorial, you'll learn how to take theexisting NFT contract you've been working with and modify it to meet some of the most common needs in the ecosystem. This includes:

- Lazy Minting NFTs
- Creating Collections
- Restricting Minting Access
- Highly Optimizing Storage
- Hacking Enumeration Methods

Introduction

Now that you have a deeper understanding of basic NFT smart contracts, we can start to get creative and implement more unique features. The basic contract works really well for simple use-cases but as you begin to explore the potential of NFTs, you can use it as a foundation to build upon.

A fun analogy would be that you now have a standard muffin recipe and it's now up to you to decide how to alter it to create your own delicious varieties, may I suggest blueberry perhaps.

Below we've created a few of these new varieties by showing potential solutions to the problems outlined above. As we demonstrate how to customize the basic NFT contract, we hope it activates your ingenuity thus introducing you to what's possible and helping you discover the true potential of NFTs.

NFT Collections and Series

NFT Collections help solve two common problems when dealing with the basic NFT contract:

- Storing repeated data.
- Organizing data and code.

The concept of a collection in the NFT space has a very loose meaning and can be interpreted in many different ways. In our case, we'll define a collection as a set of tokens that sharesimilar metadata . For example, you could create a painting and want 100 identical copies to be put for sale. In this case, all one hundred pieces would be part of the samecollection . Each piece would have the same artist, title, description, media etc.

One of the biggest problems with the basic NFT contract is that you store similar data many times. If you mint NFTs, the contract will store the metadata individually forevery single token ID . We can fix this by introducing the idea of a series, or collection, of NFTs.

A series can be thought of as a bucket of token IDs thatall share similar information. This information is specified when the series iscreated and can be the metadata, royalties, price etc. Rather than storing this information forevery token ID , you can simply store it once in the series and then associate token IDs with their respective buckets.

Restricted Access

Currently, the NFT contract allows anyone to mint NFTs. While this works well for some projects, the vast majority of dApps and creators want to restrict who can create NFTs on the contract. This is why you'll introduce an allowlist functionality for both series and for NFTs. You'll have two data structures customizable by the contract owner:

- Approved Minters
- Approved Creators

If you're an approved minter, you can freely mint NFTs for any given series. You cannot, however, create new series.

On the other hand, you can also be an approved creator. This allows you to define new series that NFTs can be minted from. It's important to note that if you're an approved creator, you're not automatically an approved minter as well. Each of these permissions need to be given by the owner of the contract and they can be revoked at any time.

Lazy Minting

Lazy minting allows users to minton demand . Rather than minting all the NFTs and spending NEAR on storage, you can instead mint the tokenswhen they are purchased . This helps to avoid burning unnecessary Gas and saves on storage for when not all the NFTs are purchased. Let's look at a common scenario to help solidify your understanding:

Benji has created an amazing digital painting of the famous Go Team gif. He wants to sell 1000 copies of it for 1 NEAR each. Using the traditional approach, he would have to mint each copy individually and pay for the storage himself. He would then need to either find or deploy a marketplace contract and pay for the storage to put 1000 copies up for sale. He

would need to burn Gas putting each token ID up for sale 1 by 1.

After that, people would purchase the NFTs, and there would be no guarantee that all or even any would be sold. There's a real possibility that nobody buys a single piece of his artwork, and Benji spent all that time, effort and money on nothing. ☹

Lazy minting would allow the NFTs to beautomatically minted on-demand . Rather than having to purchase NFTs from a marketplace, Benji could specify a price on the NFT contract and a user could directly call thenft_mint function whereby the funds would be distributed to Benji's account directly.

Using this model, NFTs wouldonly be minted when they're actually purchased and there wouldn't be any upfront fee that Benji would need to pay in order to mint all 1000 NFTs. In addition, it removes the need to have a separate marketplace contract.

With this example laid out, a high level overview of lazy minting is that it gives the ability for someone to mint "on-demand" - they're lazily minting the NFTs instead of having to mint everything up-front even if they're unsure if there's any demand for the NFTs. With this model, you don't have to waste Gas or storage fees because you're only ever minting when someone actually purchases the artwork.

New Contract File Structure

Let's now take a look at how we've implemented solutions to the issues we've discussed so far.

In your locally cloned example of thenft-tutorial check out themain branch and be sure to pull the most recent version.

git checkout main && git pull You'll notice that there's a folder at the root of the project callednft-series . This is where the smart contract code lives. If you open thesrc folder, it should look similar to the following:

src ├── approval.rs ├── enumeration.rs ├── events.rs ├── internal.rs ├── lib.rs ├── metadata.rs ├── nft_core.rs ├── owner.rs ├── royalty.rs ├── series.rs

Differences

You'll notice that most of this code is the same, however, there are a few differences between this contract and the basic NFT contract.

Main Library File

Starting withlib.rs , you'll notice that the contract struct has been modified to now store the following information.

pub owner_id: AccountId, + pub approved_minters: LookupSet, + pub approved_creators: LookupSet, pub tokens_per_owner: LookupMap>, pub tokens_by_id: UnorderedMap, - pub token_metadata_by_id: UnorderedMap, + pub series_by_id: UnorderedMap, pub metadata: LazyOption, As you can see, we've replacedtoken_metadata_by_id withseries_by_id and added two lookup sets:

- series_by_id
- : Map a series ID (u64) to its Series object.
- approved_minters
- : Keeps track of accounts that can call thenft_mint
- function.
- approved_creators
- : Keeps track of accounts that can create new series.

Series Object

In addition, we're now keeping track of a new object called aSeries .

pub

struct

Series

{ // Metadata including title, num copies etc.. that all tokens will derive from metadata :

TokenMetadata , // Royalty used for all tokens in the collection royalty :

Option < HashMap < AccountId ,

u32

, // Set of tokens in the collection tokens :

UnorderedSet < TokenId

, // What is the price of each token in this series? If this is specified, when minting, // Users will need to attach enough NEAR to cover the price. price :

Option < Balance

, // Owner of the collection owner_id :

AccountId , } This object stores information that each token will inherit from. This includes:

- Themetadata
- .
- Theroyalties
- .
- The price.

caution If a price is specified, there will be no restriction on who can mint tokens in the series. In addition, if thecopies field is specified in the metadata,only that number of NFTs can be minted. If the field is omitted, an unlimited amount of tokens can be minted. We've also added a fieldtokens which keeps track of all the token IDs that have been minted for this series. This allows us to deal with the potentialcopies cap by checking the length of the set. It also allows us to paginate through all the tokens in the series.

Creating Series

series.rs is a new file that replaces the oldminting logic. This file has been created to combine both the series creation and minting logic into one.

nft-series/src/series.rs loading ... See full example on GitHub The function takes in a series ID in the form of au64 , the metadata, royalties, and the price for tokens in the series. It will then create theSeries object and insert it into the contract's series_by_id data structure. It's important to note that the caller must be an approved creator and they must attach enough NEAR to cover storage costs.

Minting NFTs

Next, we'll look at the minting function. If you remember from before, this used to take the following parameters:

- Token ID
- Metadata
- Receiver ID
- Perpetual Royalties

With the new and improved minting function, these parameters have been changed to just two:

- The series ID
- The receiver ID.

The mint function might look complicated at first but let's break it down to understand what's happening. The first thing it does is get theseries object from the specified series ID. From there, it will check that the number of copies won't be exceeded if one is specified in the metadata.

It will then store the token information on the contract as explained in theminting section of the tutorial and map the token ID to the series. Once this is finished, a mint log will be emitted and it will ensure that enough deposit has been attached to the call. This amount differs based on whether or not the series has a price.

Required Deposit

As we went over in theminting section of this tutorial, all information stored on the contract costs NEAR. When minting, there is a required deposit to pay for this storage. Forthis contract , a series price can also be specified by the owner when the series is created. This price will be used forall NFTs in the series when they are minted. If the price is specified, the deposit must cover both the storage as well as the price.

If a priceis specified and the user more deposit than what is necessary, the excess is sent to theseries owner . There is alsono restriction on who can mint tokens for series that have a price. The caller doesnot need to be an approved minter.

Ifno price was specified in the series and the user attaches more deposit than what is necessary, the excess isrefunded to them . In addition, the contract makes sure that the caller is an approved minter in this case.

info Notice how the token ID isn't required? This is because the token ID is automatically generated when minting. The ID

stored on the contract is{series_id}:{token_id} where the token ID is a nonce that increases each time a new token is minted in a series. This not only reduces the amount of information stored on the contract but it also acts as a way to check the specific edition number. nft-series/src/series.rs loading ... See full example on GitHub

View Functions

Now that we've introduced the idea of series, more view functions have also been added.

info Notice how we've also created a new structJsonSeries instead of returning the regularSeries struct. This is because theSeries struct contains anUnorderedSet which cannot be serialized.

The common practice is to return everythingexcept theUnorderedSet in a separate struct and then have entirely different methods for accessing the data from theUnorderedSet itself. nft-series/src/enumeration.rs loading ... See full example on GitHub The view functions are listed below.

- get_series_total_supply
- : Get the total number of series currently on the contract.* Arguments: None.
- get_series
- : Paginate through all the series in the contract and return a vector ofJsonSeries
- objects.* Arguments:from_index: String | null
-
  - ,limit: number | null
-
  - .
- get_series_details
- : Get theJsonSeries
- details for a specific series.* Arguments:id: number
-
  - .
- nft_supply_for_series
- : View the total number of NFTs minted for a specific series.* Arguments:id: number
-
  - .
- nft_tokens_for_series
- : Paginate through all NFTs for a specific series and return a vector ofJsonToken
- objects.* Arguments:id: number
-
  - ,from_index: String | null
-
  - ,limit: number | null
-
  - .

info Notice how with every pagination function, we've also included a getter to view the total supply. This is so that you can use thefrom_index andlimit parameters of the pagination functions in conjunction with the total supply so you know where to end your pagination.

Modifying View Calls for Optimizations

Storing information on-chain can be very expensive. As you level up in your smart contract development skills, one area to look into is reducing the amount of information stored. View calls are a perfect example of this optimization.

For example, if you wanted to relay the edition number for a given NFT in its title, you don't necessarily need to store this on-chain for every token. Instead, you could modify the view functions to manually append this information to the title before returning it.

To do this, here's a way of modifying thenft_token function as it's central to all enumeration methods.

nft-series/src/nft_core.rs loading ... See full example on GitHubFor example if a token had a title"My Amazing Go Team Gif" and the NFT was edition 42, the new title returned would be"My Amazing Go Team Gif - 42" . If the NFT didn't have a title in the metadata, the series and edition number would be returned in the form ofSeries {} : Edition {} .

While this is a small optimization, this idea is extremely powerful as you can potentially save on a ton of storage. As an example: most of the time NFTs don't utilize the following fields in their metadata.

- issued_at
- expires_at
- starts_at
- updated_at

As an optimization, you could change the token metadata that'sstored on the contract to not include these fields but then when returning the information innft_token , you could simply add them in asnull values.

Owner File

The last file we'll look at is the owner file found atowner.rs . This file simply contains all the functions for getting and setting approved creators and approved minters which can only be called by the contract owner.

info There are some other smaller changes made to the contract that you can check out if you'd like. The most notable are:

- TheToken
- andJsonToken
- objects have beenchanged
- to reflect the new series IDs.
- All references totoken_metadata_by_id
- have beenchanged
- totokens_by_id
- Royalty functionsnow
- calculate the payout objects by using the series' royalties rather than the token's royalties.

Building the Contract

Now that you hopefully have a good understanding of the contract, let's get started building it. Run the following build command to compile the contract to wasm.

yarn build This should create a new wasm file in theout/series.wasm directory. This is what you'll be deploying on-chain.

Deployment and Initialization

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

export NFT_CONTRACT_ID= near create-account NFT_CONTRACT_ID --useFaucet near deploy NFT_CONTRACT_ID out/series.wasm Check if this worked correctly by echoing the environment variable.

echo NFT_CONTRACT_ID This should return your . The next step is to initialize the contract with some default metadata.

near call NFT_CONTRACT_ID new_default_meta '{"owner_id": "'NFT_CONTRACT_ID'"}' --accountId NFT_CONTRACT_ID If you now query for the metadata of the contract, it should return our default metadata.

near view NFT_CONTRACT_ID nft_metadata

Creating The Series

The next step is to create two different series. One will have a price for lazy minting and the other will simply be a basic series with no price. The first step is to create an ownersub-account that you can use to create both series

near create-account owner.NFT_CONTRACT_ID --masterAccount NFT_CONTRACT_ID --initialBalance 3 && export SERIES_OWNER=owner.NFT_CONTRACT_ID

Basic Series

You'll now need to create the simple series with no price and no royalties. If you try to run the following command before adding the owner account as an approved creator, the contract should throw an error.

near call NFT_CONTRACT_ID create_series '{"id": 1, "metadata": {"title": "SERIES!", "description": "testing out the new series contract", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}}' --accountId SERIES_OWNER --amount 1 The expected output is an error thrown:ExecutionError: 'Smart contract panicked: only approved creators can add a type . If you now add the series owner as a creator, it should work.

near call NFT_CONTRACT_ID add_approved_creator '{"account_id": "'SERIES_OWNER'"}' --accountId NFT_CONTRACT_ID near call NFT_CONTRACT_ID create_series '{"id": 1, "metadata": {"title": "SERIES!", "description": "testing out the new series contract", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}}' --accountId SERIES_OWNER --amount 1 If you now query for the series information, it should work!

near view NFT_CONTRACT_ID get_series Which should return something similar to:

[{ series_id: 1, metadata: { title: 'SERIES!', description: 'testing out the new series contract', 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 }, royalty: null, owner_id: 'owner.nft_contract.testnet' }]

Series With a Price

Now that you've created the first, simple series, let's create the second one that has a price of 1 NEAR associated with it.

near call NFT_CONTRACT_ID create_series '{"id": 2, "metadata": {"title": "COMPLEX SERIES!", "description": "testing out the new contract with a complex series", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}, "price": "500000000000000000000000"}' --accountId SERIES_OWNER --amount 1 If you now paginate through the series again, you should see both appear.

near view NFT_CONTRACT_ID get_series Which has

[{ series_id: 1, metadata: { title: 'SERIES!', description: 'testing out the new series contract', 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 }, royalty: null, owner_id: 'owner.nft_contract.testnet' }, { series_id: 2, metadata: { title: 'COMPLEX SERIES!', description: 'testing out the new contract with a complex series', 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 }, royalty: null, owner_id: 'owner.nft_contract.testnet' }]

Minting NFTs

Now that you have both series created, it's time to now mint some NFTs. You can either login with an existing NEAR wallet usingnear login or you can create a sub-account of the NFT contract. In our case, we'll use a sub-account.

near create-account buyer.NFT_CONTRACT_ID --masterAccount NFT_CONTRACT_ID --initialBalance 1 && export BUYER_ID=buyer.NFT_CONTRACT_ID

Lazy Minting

The first workflow you'll test out islazy minting NFTs. If you remember, the second series has a price associated with it of 1 NEAR. This means that there are no minting restrictions and anyone can try and purchase the NFT. Let's try it out.

In order to view the NFT in the NEAR wallet, you'll want thereceiver_id to be an account you have currently available in the wallet site. Let's export it to an environment variable. Run the following command but replaceYOUR_ACCOUNT_ID_HERE with your actual NEAR account ID.

export NFT_RECEIVER_ID=YOUR_ACCOUNT_ID_HERE Now if you try and run the mint command but don't attach enough NEAR, it should throw an error.

near call NFT_CONTRACT_ID nft_mint '{"id": "2", "receiver_id": "'NFT_RECEIVER_ID'"}' --accountId BUYER_ID Run the command again but this time, attach 1.5 NEAR.

near call NFT_CONTRACT_ID nft_mint '{"id": "2", "receiver_id": "'NFT_RECEIVER_ID'"}' --accountId BUYER_ID --amount 0.6 This should output the following logs.

Receipts: BrJLxCVmxLk3yNFVnwzpjZPDRhiCinNinLQwj9A7184P, 3UwUgdq7i1VpKyw3L5bmJvbUiqvFRvpi2w7TfqmnPGH6 Log [nft_contract.testnet]: EVENT_JSON: {"standard":"nep171","version":"nft-1.0.0","event":"nft_mint","data":[{"owner_id":"benjiman.testnet","token_ids":["2:1"]}]} Transaction Id FxWLFGuap7SFrUPLskVr7Uxxq8hpDtAG76AvshWppBVC To see the transaction in the transaction explorer, please open this url in your browser https://testnet.nearblocks.io/txns/FxWLFGuap7SFrUPLskVr7Uxxq8hpDtAG76AvshWppBVC " If you check the explorer link, it should show that the owner received on the order of0.59305 NEAR .

Becoming an Approved Minter

If you try to mint the NFT for the simple series with no price, it should throw an error saying you're not an approved minter.

near call NFT_CONTRACT_ID nft_mint '{"id": "1", "receiver_id": "'NFT_RECEIVER_ID'"}' --accountId BUYER_ID --amount 0.1 Go ahead and run the following command to add the buyer account as an approved minter.

near call NFT_CONTRACT_ID add_approved_minter '{"account_id": "'BUYER_ID'"}' --accountId NFT_CONTRACT_ID If you now run the mint command again, it should work.

near call NFT_CONTRACT_ID nft_mint '{"id": "1", "receiver_id": "'NFT_RECEIVER_ID'"}' --accountId BUYER_ID --amount

0.1

Viewing the NFTs in the Wallet

Now that you've received both NFTs, they should show up in the NEAR wallet. Open the collectibles tab and search for the contract with the titleNFT Series Contract and you should own two NFTs. One should be the complex series and the other should just be the simple version. Both should have- 1 appended to the end of the title because the NFTs are the first editions for each series.

Hurray! You've successfully deployed and tested the series contract!GO TEAM! .

Conclusion

In this tutorial, you learned how to take the basic NFT contract and iterate on it to create a complex and custom version to meet the needs of the community. You optimized the storage, introduced the idea of collections, created a lazy minting functionality, hacked the enumeration functions to save on storage, and created an allowlist functionality.

You then built the contract and deployed it on chain. Once it was on-chain, you initialized it and created two sets of series. One was complex with a price and the other was a regular series. You lazy minted an NFT and purchased it for1.5 NEAR and then added yourself as an approved minter. You then minted an NFT from the regular series and viewed them both in the NEAR wallet.

Thank you so much for going through this journey with us! I wish you all the best and am eager to see what sorts of neat and unique use-cases you can come up with. If you have any questions, feel free to ask on ourDiscord or any other social media channels we have. If you run into any issues or have feedback, feel free to use theFeedback button on the right.

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 Marketplace Next Crossword Game Overview