Approvals

In this tutorial you'll learn the basics of an approval management system which will allow you to grant others access to transfer NFTs on your behalf. This is the backbone of all NFT marketplaces and allows for some complex yet beautiful scenarios to happen. If you're joining us for the first time, feel free to clonethis repository and checkout the4.core branch to follow along.

git checkout 4.core tip If you wish to see the finished code for thisApproval tutorial, you can find it on the5.approval branch.

Introduction

Up until this point you've created a smart contract that allows users to mint and transfer NFTs as well as query for information using theenumeration standard . As we've been doing in the previous tutorials, let's break down the problem into smaller, more digestible, tasks. Let's first define some of the end goals that we want to accomplish as per theapproval management extension of the standard. We want a user to have the ability to:

- Grant other accounts access to transfer their NFTs on a per token basis.
- Check if an account has access to a specific token.
- Revoke a specific account the ability to transfer an NFT.
- Revokeall
- other accounts the ability to transfer an NFT.

If you look at all these goals, they are all on a per token basis. This is a strong indication that you should change theToken struct which keeps track of information for each token.

Allow an account to transfer your NFT

Let's start by trying to accomplish the first goal. How can you grant another account access to transfer an NFT on your behalf?

The simplest way that you can achieve this is to add a list of approved accounts to theToken struct. When transferring the NFT, if the caller is not the owner, you could check if they're in the list.

Before transferring, you would need to clear the list of approved accounts since the new owner wouldn't expect the accounts approved by the original owner to still have access to transfer their new NFT.

The problem

On the surface, this would work, but if you start thinking about the edge cases, some problems arise. Often times when doing development, a common approach is to think about the easiest and most straightforward solution. Once you've figured it out, you can start to branch off and think about optimizations and edge cases.

Let's consider the following scenario. Benji has an NFT and gives two separate marketplaces access to transfer his token. By doing so, he's putting the NFT for sale (more about that in themarketplace integrations section). Let's say he put the NFT for sale for 1 NEAR on both markets. The tokens list of approved account IDs would look like the following:

Token: { owner_id: Benji approved_accounts_ids: [marketplace A, marketplace B] } Josh then comes along and purchases the NFT on marketplace A for 1 NEAR. This would take the sale down from the marketplace A and clear the list of approved accounts. Marketplace B, however, still has the token listed for sale for 1 NEAR and has no way of knowing that the token was purchased on marketplace A by Josh. The new token struct would look as follows:

Token: { owner_id: Josh approved_accounts_ids: [] } Let's say Josh is low on cash and wants to flip this NFT and put it for sale for 10 times the price on marketplace B. He goes to put it for sale and for whatever reason, the marketplace is built in a way that if you try to put a token up for sale twice, it keeps the old sale data. This would mean that from marketplace B's perspective, the token is still for sale for 1 NEAR (which was the price that Benji had originally listed it for).

Since Josh approved the marketplace to try and put it for sale, the token struct would look as follows:

Token: { owner_id: Josh approved_accounts_ids: [marketplace A, marketplace B] } If Mike then comes along and purchases the NFT for only 1 NEAR on marketplace B, the marketplace would go to try and transfer the NFT and since technically, Josh approved the marketplace and it's in the list of approved accounts, the transaction would go through properly.

The solution

Now that we've identified a problem with the original solution, let's think about ways that we can fix it. What would happen now if, instead of just keeping track of a list of approved accounts, you introduced a specific ID that went along with each approved account. The new approved accounts would now be a map instead of a list. It would map an account to it'sapproval id .

For this to work, you need to make sure that the approval ID isalways a unique, new ID. If you set it as an integer that always increases by 1 whenever u approve an account, this should work. Let's consider the same scenario with the new solution.

Benji puts his NFT for sale for 1 NEAR on marketplace A and marketplace B by approving both marketplaces. The "next approval ID" would start off at 0 when the NFT was first minted and will increase from there. This would result in the following token struct:

Token: { owner_id: Benji approved_accounts_ids: { marketplace A: 0 marketplace B: 1 } next_approval_id: 2 } When Benji approved marketplace A, it took the original value ofnext_approval_id which started off at 0. The marketplace was then inserted into the map and the next approval ID was incremented. This process happened again for marketplace B and the next approval ID was again incremented where it's now 2.

Josh comes along and purchases the NFT on marketplace A for 1 NEAR. Notice how the next approval ID stayed at 2:

Token: { owner_id: Josh approved_accounts_ids: {} next_approval_id: 2 } Josh then flips the NFT because he's once again low on cash and approves marketplace B:

Token: { owner_id: Josh approved_accounts_ids: { marketplace B: 2 } next_approval_id: 3 } The marketplace is inserted into the map and the next approval ID is incremented. From marketplace B's perspective it stores it's original approval ID from when Benji put the NFT up for sale which has a value of 1. If Mike were to go and purchase the NFT on marketplace B for the original 1 NEAR sale price, the NFT contract should panic. This is because the marketplace is trying to transfer the NFT with an approval ID 1 but the token struct shows that itshould have an approval ID of 2.

Expanding theToken

andJsonToken structs

Now that you understand the proposed solution to the original problem of allowing an account to transfer your NFT, it's time to implement some of the logic. The first thing you should do is modify theToken andJsonToken structs to reflect the new changes. Let's switch over to thenft-contract/src/metadata.rs file:

nft-contract/src/metadata.rs loading ... See full example on GitHub You'll then need to initialize both theapproved_account_ids andnext_approval_id to their default values when a token is minted. Switch to thenft-contract/src/mint.rs file and when creating theToken struct to store in the contract, let's set the next approval ID to be 0 and the approved account IDs to be an empty map:

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

Approving accounts

Now that you've added the support for approved account IDs and the next approval ID on the token level, it's time to add the logic for populating and changing those fields through a function callednft_approve . This function should approve an account to have access to a specific token ID. Let's move to thenft-contract/src/approval.rs file and edit thenft_approve function:

nft-contract/src/approval.rs loading ... See full example on GitHub The function will first assert that the user has attachedat least one yoctoNEAR (which we'll implement soon). This is both for security and to cover storage. When someone approves an account ID, they're storing that information on the contract. As you saw in theminting tutorial , you can either have the smart contract account cover the storage, or you can have the users cover that cost. The latter is more scalable and it's the approach you'll be working with throughout this tutorial.

After the assertion comes back with no problems, you get the token object and make sure that only the owner is calling this method. Only the owner should be able to allow other accounts to transfer their NFTs. You then get the next approval ID and insert the passed in account into the map with the next approval ID. If it's a new approval ID, storage must be paid. If it's not a new approval ID, no storage needs to be paid and only attaching 1 yoctoNEAR would be enough.

You then calculate how much storage is being used by adding that new account to the map and increment the tokensnext_approval_id by 1. After inserting the token object back into thetokens_by_id map, you refund any excess storage.

You'll notice that the function contains an optionalmsg parameter. This message is actually the foundation of all NFT marketplaces on NEAR.

Marketplace Integrations

If a message was provided into the function, you're going to perform a cross contract call to the account being given access. This cross contract call will invoke thenft_on_approve function which will parse the message and act accordingly. Let's consider a general use case.

We have a marketplace that expects it's sale conditions to be passed in through the message field. Benji approves the marketplace with thenft_approve function and passes in a stringified JSON to the message which will outline sale conditions. These sale conditions could look something like the following:

sale_conditions: { price: 5 } By leaving the message field type as just a string, this generalizes the process and allows users to input sale conditions for many different marketplaces. It is up to the person approving to pass in an appropriate message that the marketplace can properly decode and use. This is usually done through the marketplace's frontend app which would know how to construct themsg in a useful way.

Internal functions

Now that the core logic for approving an account is finished, you need to implement theassert_at_least_one_yocto andbytes_for_approved_account functions. Move to thenft-contract/src/internal.rs file and copy the following function right below theassert_one_yocto function.

nft-contract/src/internal.rs loading ... See full example on GitHub Next, you'll need to copy the logic for calculating how many bytes it costs to store an account ID. Place this function at the very top of the page:

nft-contract/src/internal.rs loading ... See full example on GitHub Now that the logic for approving accounts is finished, you need to change the restrictions for transferring.

Changing the restrictions for transferring NFTs

Currently, an NFT canonly be transferred by its owner. You need to change that restriction so that people that have been approved can also transfer NFTs. In addition, you'll make it so that if an approval ID is passed, you can increase the security and check if both the account trying to transfer is in the approved listand they correspond to the correct approval ID. This is to address the problem we ran into earlier.

In theinternal.rs file, you need to change the logic of theinternal_transfer method as that's where the restrictions are being made. Change the internal transfer function to be the following:

nft-contract/src/internal.rs loading ... See full example on GitHub This will check if the sender isn't the owner and then if they're not, it will check if the sender is in the approval list. If an approval ID was passed into the function, it will check if the sender's actual approval ID stored on the contract matches the one passed in.

Refunding storage on transfer

While you're in the internal file, you're going to need to add methods for refunding users who have paid for storing approved accounts on the contract when an NFT is transferred. This is because you'll be clearing theapproved_account_ids map whenever NFTs are transferred and so the storage is no longer being used.

Right below thebytes_for_approved_account_id function, copy the following two functions:

nft-contract/src/internal.rs loading ... See full example on GitHub These will be useful in the next section where you'll be changing thenft_core functions to include the new approval logic.

Changes tonft_core.rs

Head over to thenft-contract/src/nft_core.rs file and the first change that you'll want to make is to add anapproval_id to both thenft_transfer andnft_transfer_call functions. This is so that anyone trying to transfer the token that isn't the owner must pass in an approval ID to address the problem seen earlier. If they are the owner, the approval ID won't be used as we saw in theinternal_transfer function.

nft-contract/src/nft_core.rs loading ... See full example on GitHub You'll then need to add anapproved_account_ids map to the parameters ofnft_resolve_transfer . This is so that you can refund the list if the transfer went through properly.

nft-contract/src/nft_core.rs loading ... See full example on GitHub Moving over tonft_transfer , the only change that you'll need to make is to pass in the approval ID into theinternal_transfer function and then refund the previous tokens approved account IDs after the transfer is finished

nft-contract/src/nft_core.rs loading ... See full example on GitHub Next, you need to do the same tonft_transfer_call but instead of refunding immediately, you need to attach the previous token's approved account IDs tonft_resolve_transfer instead as there's still the possibility that the transfer gets reverted.

nft-contract/src/nft_core.rs loading ... See full example on GitHub You'll also need to add the tokens approved account IDs to theJsonToken being returned bynft_token .

nft-contract/src/nft_core.rs loading ... See full example on GitHub Finally, you need to add the logic for refunding the

approved account IDs innft_resolve_transfer . If the transfer went through, you should refund the owner for the storage being released by resetting the tokensapproved_account_ids field. If, however, you should revert the transfer, it wouldn't be enough to just not refund anybody. Since the receiver briefly owned the token, they could have added their own approved account IDs and so you should refund them if they did so.

nft-contract/src/nft_core.rs loading ... See full example on GitHubWith that finished, it's time to move on and complete the next task.

Check if an account is approved

Now that the core logic is in place for approving and refunding accounts, it should be smooth sailing from this point on. You now need to implement the logic for checking if an account has been approved. This should take an account and token ID as well as an optional approval ID. If no approval ID was provided, it should simply return whether or not the account is approved.

If an approval ID was provided, it should return whether or not the account is approved and has the same approval ID as the one provided. Let's move to thenft-contract/src/approval.rs file and add the necessary logic to thenft_is_approved function.

nft-contract/src/approval.rs loading ... See full example on GitHubLet's now move on and add the logic for revoking an account

Revoke an account

The next step in the tutorial is to allow a user to revoke a specific account from having access to their NFT. The first thing you'll want to do is assert one yocto for security purposes. You'll then need to make sure that the caller is the owner of the token. If those checks pass, you'll need to remove the passed in account from the tokens approved account IDs and refund the owner for the storage being released.

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

Revoke all accounts

The final step in the tutorial is to allow a user to revoke all accounts from having access to their NFT. This should also assert one yocto for security purposes and make sure that the caller is the owner of the token. You then refund the owner for releasing all the accounts in the map and then clear theapproved_account_ids .

nft-contract/src/approval.rs loading ... See full example on GitHubWith that finished, it's time to deploy and start testing the contract.

Testing the new changes

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, it's best practice to create a new account and deploy the contract there.

Deployment

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

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

yarn build && near deploy APPROVAL_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 APPROVAL_NFT_CONTRACT_ID new_default_meta '{"owner_id": "'APPROVAL_NFT_CONTRACT_ID'"}' --accountId APPROVAL_NFT_CONTRACT_ID Next, you'll need to mint a token. By running this command, you'll mint a token with a token ID"approval-token" and the receiver will be your new account.

near call APPROVAL_NFT_CONTRACT_ID nft_mint '{"token_id": "approval-token", "metadata": {"title": "Approval Token", "description": "testing out the new approval extension of the standard", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmaqyhrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}, "receiver_id": "'APPROVAL_NFT_CONTRACT_ID'"}' --accountId APPROVAL_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 APPROVAL_NFT_CONTRACT_ID nft_tokens_for_owner '{"account_id": "'APPROVAL_NFT_CONTRACT_ID'", "limit": 10}' This should return an output similar to the following:

[{ "token_id": "approval-token", "owner_id": "approval.goteam.examples.testnet", "metadata": { "title": "Approval Token", "description": "testing out the new approval 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": {} }] Notice how the approved account IDs are now being returned from the function? This is a great sign! You're now ready to move on and approve an account to have access to your token.

Approving an account

At this point, you should have two accounts. One stored underNFT_CONTRACT_ID and the other under theAPPROVAL_NFT_CONTRACT_ID environment variable. You can use both of these accounts to test things out. If you approve your old account, it should have the ability to transfer the NFT to itself.

Execute the following command to approve the account stored underNFT_CONTRACT_ID to have access to transfer your NFT with an ID"approval-token" . You don't need to pass a message since the old account didn't implement thenft_on_approve function. In addition, you'll need to attach enough NEAR to cover the cost of storing the account on the contract. 0.1 NEAR should be more than enough and you'll be refunded any excess that is unused.

near call APPROVAL_NFT_CONTRACT_ID nft_approve '{"token_id": "approval-token", "account_id": "'NFT_CONTRACT_ID'"}' --accountId APPROVAL_NFT_CONTRACT_ID --deposit 0.1 If you call the same enumeration method as before, you should see the new approved account ID being returned.

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

[{ "token_id": "approval-token", "owner_id": "approval.goteam.examples.testnet", "metadata": { "title": "Approval Token", "description": "testing out the new approval 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": { "goteam.examples.testnet": 0 } }]

Transferring an NFT as an approved account

Now that you've approved another account to transfer the token, you can test that behavior. You should be able to use the other account to transfer the NFT to itself by which the approved account IDs should be reset. Let's test transferring the NFT with the wrong approval ID:

near call APPROVAL_NFT_CONTRACT_ID nft_transfer '{"receiver_id": "'NFT_CONTRACT_ID'", "token_id": "approval-token", "approval_id": 1}' --accountId NFT_CONTRACT_ID --depositYocto 1 Example response: kind: { ExecutionError: "Smart contract panicked: panicked at 'assertion failed: (left == right)\n" + ' left: 0,\n' + " right: 1: The actual approval_id 0 is different from the given approval_id 1', src/internal.rs:165:17" }, If you pass the correct approval ID which is0 , everything should work fine.

near call APPROVAL_NFT_CONTRACT_ID nft_transfer '{"receiver_id": "'NFT_CONTRACT_ID'", "token_id": "approval-token", "approval_id": 0}' --accountId NFT_CONTRACT_ID --depositYocto 1 If you again call the enumeration method, you should see the owner updated and the approved account IDs reset.

[{ "token_id": "approval-token", "owner_id": "goteam.examples.testnet", "metadata": { "title": "Approval Token", "description": "testing out the new approval 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": {} }] Let's now test the approval ID incrementing across different owners. If you approve the account that originally minted the token, the approval ID should be 1 now.

near call APPROVAL_NFT_CONTRACT_ID nft_approve '{"token_id": "approval-token", "account_id": "'APPROVAL_NFT_CONTRACT_ID'"}' --accountId NFT_CONTRACT_ID --deposit 0.1 Calling the view function again show now return an approval ID of 1 for the account that was approved.

near view APPROVAL_NFT_CONTRACT_ID nft_tokens_for_owner '{"account_id": "'NFT_CONTRACT_ID'", "limit": 10}' Example response: [{ "token_id": "approval-token", "owner_id": "goteam.examples.testnet", "metadata": { "title": "Approval Token", "description": "testing out the new approval 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": { "approval.goteam.examples.testnet": 1 } }] With the testing finished, you've successfully implemented the approvals extension to the standard!

Conclusion

Today you went through a lot of logic to implement the approvals extension so let's break down exactly what you did.

First, you explored the basic approach of how to solve the problem. You then went through and discovered some of the problems with that solution and learned how to fix it .

After understanding what you should do to implement the approvals extension, you started to modify the JsonToken and Token structs in the contract. You then implemented the logic for approving accounts and saw how marketplaces are integrated.

After implementing the logic behind approving accounts, you went and changed the restrictions needed to transfer NFTs. The last step you did to finalize the approving logic was to go back and edit the nft_core files to be compatible with the new changes.

At this point, everything was implemented in order to allow accounts to be approved and you extended the functionality of the core standard to allow for approved accounts to transfer tokens.

You implemented a view method to check if an account is approved and to finish the coding portion of the tutorial, you implemented the logic necessary to revoke an account as well as revoke all accounts .

After this, the contract code was finished and it was time to move onto testing where you created a account and tested the approving and transferring for your NFTs.

In the next tutorial, you'll learn about the royalty standards and how you can interact with NFT marketplaces.

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
- Approval standard:NEP178
- , version1.1.0 Edit this page Last updatedonFeb 16, 2024 bygarikbesson Was this page helpful? Yes No

Previous Core Next Royalty