

Events

In this tutorial, you'll learn about the [events standard](#) and how to implement it in your smart contract.

Introduction

To get started, either switch to the `6.royalty` branch from our [GitHub repository](#) , or continue your work from the previous tutorials.

git checkout 6.royalty tip If you wish to see the finished code for this Events tutorial, you can find it on the `7.events` branch.

Understanding the use case

Have you ever wondered how the wallet knows which NFTs you own and how it can display them in the [collectibles tab](#) ? Originally, an [indexer](#) was used and it listened for any functions starting with `nft_` on your account. These contracts were then flagged on your account as likely NFT contracts.

When you navigated to your collectibles tab, the wallet would then query all those contracts for the list of NFTs you owned using the `nft_tokens_for_owner` function you saw in the [enumeration tutorial](#) .

The problem

This method of flagging contracts was not reliable as each NFT-driven application might have its own way of minting or transferring NFTs. In addition, it's common for apps to transfer or mint many tokens at a time using batch functions.

The solution

A standard was introduced so that smart contracts could emit an event anytime NFTs were transferred, minted, or burnt. This event was in the form of a log. No matter how a contract implemented the functionality, an indexer could now listen for those standardized logs.

As per the standard, you need to implement a logging functionality that gets fired when NFTs are transferred or minted. In this case, the contract doesn't support burning so you don't need to worry about that for now.

It's important to note the standard dictates that the log should begin with `"EVENT_JSON:"` . The structure of your log should, however, always contain the 3 following things:

- standard
- : the current name of the standard (e.g. `nep171`)
- version
- : the version of the standard you're using (e.g. `1.0.0`)
- event
- : a list of events you're emitting.

The event interface differs based on whether you're recording transfers or mints. The interface for both events is outlined below.

Transfer events :

- Optional
- `-authorized_id`
- : the account approved to transfer on behalf of the owner.
- `old_owner_id`
- : the old owner of the NFT.
- `new_owner_id`
- : the new owner that the NFT is being transferred to.
- `token_ids`
- : a list of NFTs being transferred.
- Optional
- `-memo`
- : an optional message to include with the event.

Minting events :

- `owner_id`
- : the owner that the NFT is being minted to.
- `token_ids`
- : a list of NFTs being transferred.
- Optional
- `-memo`
- : an optional message to include with the event.

Examples

In order to solidify your understanding of the standard, let's walk through three scenarios and see what the logs should look like.

Scenario A - simple mint

In this scenario, Benji wants to mint an NFT to Mike with a token ID `"team-token"` and he doesn't include a message. The log should look as follows.

`EVENT_JSON : { "standard" :`

`"nep171" , "version" :`

`"1.0.0" , "event" :`

```
"nft_mint" , "data" :  
[ { "owner_id" :  
"mike.testnet" ,  
"token_ids" :  
[ "team-token" ] } ] }
```

Scenario B - batch mint

In this scenario, Benji wants to perform a batch mint. He will mint an NFT to Mike, Damian, Josh, and Dorian. Dorian, however, will get two NFTs. Each token ID will be "team-token" followed by an incrementing number. The log is as follows.

```
EVENT_JSON : { "standard" :  
"nep171" , "version" :  
"1.0.0" , "event" :  
"nft_mint" , "data" :  
[ { "owner_id" :  
"mike.testnet" ,  
"token_ids" :  
[ "team-token0" ] } , { "owner_id" :  
"damian.testnet" ,  
"token_ids" :  
[ "team-token1" ] } , { "owner_id" :  
"josh.testnet" ,  
"token_ids" :  
[ "team-token2" ] } { "owner_id" :  
"dorian.testnet" ,  
"token_ids" :  
[ "team-token3" ,  
"team-token4" ] } , ] }
```

Scenario C - transfer NFTs

In this scenario, Mike is transferring both his team tokens to Josh. The log should look as follows.

```
EVENT_JSON : { "standard" :  
"nep171" , "version" :  
"1.0.0" , "event" :  
"nft_transfer" , "data" :  
[ { "old_owner_id" :  
"mike.testnet" ,  
"new_owner_id" :  
"josh.testnet" ,  
"token_ids" :  
[ "team-token" ,  
"team-token0" ] ,  
"memo" :  
"Go Team!" } ] }
```

Modifications to the contract

At this point, you should have a good understanding of what the end goal should be so let's get to work! Open the repository and create a new file in the `nft-contract/src` directory called `events.rs`. This is where your log structs will live.

Creating the events file

Copy the following into your file. This will outline the structs for your `EventLog`, `NftMintLog`, and `NftTransferLog`. In addition, we've added a way for `EVENT_JSON`: to be prefixed whenever you log the `EventLog`.

`nft-contract/src/events.rs` loading ... [See full example on GitHub](#) This requires the `serde_json` package which you can easily add to your `nft-contract/Cargo.toml` file:

`nft-contract/Cargo.toml` loading ... [See full example on GitHub](#)

Adding modules and constants

Now that you've created a new file, you need to add the module to the `lib.rs` file. In addition, you can define two constants for the standard and version that will be used across our contract.

`nft-contract/src/lib.rs` loading ... [See full example on GitHub](#)

Logging minted tokens

Now that all the tools are set in place, you can now implement the actual logging functionality. Since the contract will only be minting tokens in one place, it's trivial where you should place the log. Open the `nft-contract/src/mint.rs` file and navigate to the bottom of the file. This is where you'll construct the log for minting. Anytime someone successfully mints an NFT, it will now correctly emit a log.

`nft-contract/src/mint.rs` loading ... [See full example on GitHub](#)

Logging transfers

Let's open the `nft-contract/src/internal.rs` file and navigate to the `internal_transfer` function. This is the location where you'll build your transfer logs. Whenever an NFT is transferred, this function is called and so you'll correctly be logging the transfers.

`nft-contract/src/internal.rs` loading ... [See full example on GitHub](#) This solution, unfortunately, has an edge case which will break things. If an NFT is transferred via the `nft_transfer_call` function, there's a chance that the transfer will be reverted if the `nft_on_transfer` function returns `true`. Taking a look at the logic for `nft_transfer_call`, you can see why this is a problem.

When `nft_transfer_call` is invoked, it will:

- Call `internal_transfer`
- to perform the actual transfer logic.
- Initiate a cross-contract call and invoke the `nft_on_transfer` function.
- Resolve the promise and perform logic in `nft_resolve_transfer`
- * This will either return `true` meaning the transfer went fine or it will revert the transfer and return `false`.

If you only place the log in the `internal_transfer` function, the log will be emitted and the indexer will think that the NFT was transferred. If the transfer is reverted during `nft_resolve_transfer`, however, that event should also be emitted. Anywhere that an NFT could be transferred, we should add logs. Replace `nft_resolve_transfer` with the following code.

`nft-contract/src/nft_core.rs` loading ... [See full example on GitHub](#) In addition, you need to add an `authorized_id` and `memo` to the parameters for `nft_resolve_transfer` as shown below.

`nft-contract/src/nft_core.rs` loading ... [See full example on GitHub](#) The last step is to modify the `nft_transfer_call` logic to include these new parameters:

`nft-contract/src/nft_core.rs` loading ... [See full example on GitHub](#) With that finished, you've successfully implemented the events standard and it's time to start testing.

Deploying the contract

For the purpose of readability and ease of development, instead of redeploying the contract to the same account, let's create an account and deploy to that instead. You could have deployed to the same account as none of the changes you implemented in this tutorial would have caused errors.

Deployment

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

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

`yarn build && near deploy EVENTS_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 EVENTS_NFT_CONTRACT_ID new_default_meta '{"owner_id": "EVENTS_NFT_CONTRACT_ID"}' --accountId EVENTS_NFT_CONTRACT_ID` Next, you'll need to mint a token. By running this command, you'll mint a token with a token ID "events-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 EVENTS_NFT_CONTRACT_ID nft_mint '{"token_id": "events-token", "metadata": {"title": "Events Token", "description": "testing out the new events extension of the standard", "media": "https://bafybeiftczwrtyr3k7a2k4vutd3amkwsmagyhqrdzlhvpt33dyjivufqusq.ipfs.dweb.link/goteam-gif.gif"}, "receiver_id": "EVENTS_NFT_CONTRACT_ID"}' --accountId EVENTS_NFT_CONTRACT_ID --amount 0.1` You can check to see if everything went through properly by looking at the output in your CLI:

Doing `account.functionCall()` Receipts: F4oxNfv54cqWUwLUJ7h74H1iE66Y3H7QDfZMmGENwSxd, BJxKNFRuLDdbhbGeLA3UBSbL8UicU7oqHsWGink5WX7S Log [events.goteam.examples.testnet]: EVENT_JSON: {"standard":"nep171","version":"1.0.0","event":"nft_mint","data":{"owner_id":"events.goteam.examples.testnet","token_ids":["events-token"]}} Transaction Id 4Wy2KQVTuAWQHw5jXcRAbrz7bNyZBoiPEvLcGougciyk To see the transaction in the transaction explorer, please open this url in your browser <https://testnet.nearblocks.io/txns/4Wy2KQVTuAWQHw5jXcRAbrz7bNyZBoiPEvLcGougciyk> " You can see that the event was properly logged!

Transferring

You can now test if your transfer log works as expected by sending `benjiman.testnet` your NFT.

near call EVENTS_NFT_CONTRACT_ID nft_transfer '{"receiver_id": "benjiman.testnet", "token_id": "events-token", "memo": "Go Team :)", "approval_id": 0}' --accountId EVENTS_NFT_CONTRACT_ID --depositYocto 1 This should return an output similar to the following:

Doing `account.functionCall()` Receipts: EoqBxrpv9Dgb8KqK4FdeREawVVLWepEUR15KPNuZ4fGD, HZ4xQpbgc8EfU3PiV72Lvfb2f3dVC1n9aVTbQds9zfR Log [events.goteam.examples.testnet]: Memo: Go Team :) Log [events.goteam.examples.testnet]: EVENT_JSON: {"standard":"nep171","version":"1.0.0","event":"nft_transfer","data":{"authorized_id":"events.goteam.examples.testnet","old_owner_id":"events.goteam.examples.testnet","new_owner_id":"benjiman.testnet","token_ids":["events-token"],"memo":"Go Team :)"}} Transaction Id 4S1VrepKzA6HxvPj3cK12vaT7Dt4vxJRWESA1ym1xdvH To see the transaction in the transaction explorer, please open this url in your browser <https://testnet.nearblocks.io/txns/4S1VrepKzA6HxvPj3cK12vaT7Dt4vxJRWESA1ym1xdvH> " Hurray! At this point, your NFT contract is fully complete and the events standard has been implemented.

Conclusion

Today you went through the [events standard](#) and implemented the necessary logic in your smart contract. You created events for [minting](#) and [transferring](#) NFTs. You then deployed and [tested](#) your changes by minting and transferring NFTs.

In the next tutorial, you'll look at the basics of a marketplace contract and how it was built.

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
- Events standard:[NEP297 extension](#)
- , version1.1.0 [Edit this page](#) Last updated on Feb 16, 2024 by garikbesson Was this page helpful? Yes No

[Previous](#) [Royalty](#) [Next](#) [Marketplace](#)