Building an NFT indexer

Source code for the tutorial near-examples/near-lake-nft-indexer : source code for this tutorial

The End

This tutorial ends with a working NFT indexer built on topNEAR Lake Framework JS . The indexer is watching fornft_mint Events and prints some relevant data:

- receiptId
- of theReceipt
- where the mint has happened
- Marketplace
- NFT owner account name
- Links to the NFTs on the marketplaces

The final source code is available on the GitHubnear-examples/near-lake-nft-indexer

Motivation

NEAR Protocol had introduced a nice featureEvents . The Events allow a contract developer to add standardized logs to theExecutionOutcomes thus allowing themselves or other developers to read those logs in more convenient manner via API or indexers.

The Events have a fieldstandard which aligns with NEPs. In this tutorial we'll be talking aboutNEP-171 Non-Fungible Token standard .

In this tutorial our goal is to show you how you can "listen" to the Events contracts emit and how you can benefit from them.

As the example we will be building an indexer that watches all the NFTs minted following theNEP-171 Events standard, assuming we're collectors who don't want to miss a thing. Our indexer should notice every single NFT minted and give us a basic set of data like: in what Receipt it was minted, and show us the link to a marketplace (we'll coverParas andMintbase in our example).

We will use JS version ofNEAR Lake Framework in this tutorial. Though the concept is the same for Rust, but we want to show more people that it's not that complex to build your own indexer.

Preparation

Credentials Please, ensure you've the credentials set up as described on theCredentials page. Otherwise you won't be able to get the code working. You will need:

- node
- installed and configured

Let's create our project folder

mkdir lake-nft-indexer && cd lake-nft-indexer Let's add thepackage.json

{ "name": "lake-nft-indexer", "version": "1.0.0", "description": "", "main": "index.js", "scripts": { "start": "tsc && node index.js" }, "dependencies": { "near-lake-framework": "^1.0.2" }, "devDependencies": { "typescript": "^4.6.4" } } You may have noticed we've addedtypescript as a dev dependency. Let's configure the TypeScript. We'll need to createtsconfig.json file for that

{ "compilerOptions": { "lib": ["ES2019", "dom"] } } ES2019 edition Please, note theES2019 edition used. We require it because we are going to use.flatMap() and.flat() in our code. These methods were introduces inES2019 . Though you can use even more recent Let's create emptyindex.ts in the project root and thus finish the preparations.

npm install Now we can start a real work.

Set up NEAR Lake Framework

In theindex.ts let's importstartStream function andtypes fromnear-lake-framework :

import

{ startStream , types }

from

'near-lake-framework' ; Add the instantiation ofLakeConfig below:

const lakeConfig : types . LakeConfig

=

{ s3BucketName :

"near-lake-data-mainnet" , s3RegionName :

"eu-central-1" , startBlockHeight :

66264389 , } ; Just a few words on the config, we have sets3BucketName for mainnet, defaults3RegionName and a fresh-ish block height forstartBlockHeight . You can go toNEAR Explorer and getthe freshest block height for your setup. Though you can use the same as we do.

Now we need to create a callback function that we'll be called to handleStreamerMessage our indexer receives.

async

function

handleStreamerMessage (streamerMessage : types . StreamerMessage) :

Promise < void

{

} Callback function requirements Innear-lake-framework JS library the handler have to be presented as a callback function. This function have to:

- be asynchronous
- accept an argument of typeStreamerMessage
- return nothing (void
-) And an actual start of our indexer in the end of theindex.ts

(async

( )

=>

{ await

startStream (lakeConfig , handleStreamerMessage) ; }) ( ) ; The finalindex.ts at this moment should look like the following:

import

{ startStream , types }

from

'near-lake-framework' ;

const lakeConfig : types . LakeConfig

=

{ s3BucketName :

"near-lake-data-mainnet" , s3RegionName :

"eu-central-1" , startBlockHeight :

66264389 , } ;

async

function

handleStreamerMessage (streamerMessage : types . StreamerMessage) :

Promise < void

{

}

(async

( )

=>

{ await

startStream (lakeConfig , handleStreamerMessage) ; }) ( ) ;

Events and where to catch them

First of all let's find out where we can catch the Events. We hope you are familiar with how the Data Flow in NEAR Blockchain , but let's revise our knowledge:

- Mint an NFT is an action in an NFT contract (doesn't matter which one)
- Actions are located in a Receipt
- A result of the Receipt execution is ExecutionOutcome
- ExecutionOutcome
- in turn, catches the logs a contract "prints"
- Events
- built on top of the logs

This leads us to the realization that we can watch only for ExecutionOutcomes and ignore everything elseStreamerMessage brings us.

Also, we need to define an interface to catch the Events. Let's copy the interface definition from the Events Nomicon page and paste it before thehandleStreamerMessage function.

interface

EventLogData

{ standard :

string , version :

string , event :

string , data ? :

unknown , } ;

Catching only the data we need

Inside the callback functionhandleStreamerMessage we've prepared in the Preparation section let's start filtering the data we need:

async

function

handleStreamerMessage (streamerMessage : types . StreamerMessage) :

Promise < void

{ const relevantOutcomes = streamerMessage . shards . flatMap (shard => shard . receiptExecutionOutcomes)

} We have iterated through all the Shards and collected the lists of all ExecutionOutcomes into a single list (in our case we don't care on which Shard did the mint happen)

Now we want to deal only with those ExecutionOutcomes that contain logs of Events format. Such logs start withEVENT_JSON: according to the Events docs .

Also, we don't require all the data from ExecutionOutcome, let's handle it:

async

function

handleStreamerMessage (streamerMessage : types . StreamerMessage) :

Promise < void

{ const relevantOutcomes = streamerMessage . shards . flatMap (shard => shard . receiptExecutionOutcomes) . map (outcome =>

({ receipt :

{ id : outcome . receipt . receiptId , receiverId : outcome . receipt . receiverId , } , events : outcome . executionOutcome . outcome . logs . map ((log :

string) :

EventLogData

=>

{ const

[_ , probablyEvent]

= log . match (/ ^ EVENT_JSON: (. *)

/)

??

[ ] try

{ return

JSON . parse (probablyEvent) }

catch

(e)

{ return } }) . filter (event => event !==

undefined) }))

} Let us explain what we are doing here:

1. We are walking through the ExecutionOutcomes
2. We are constructing a list of objects containingreceipt
3. (it's id and the receiver) andevents
4. containing the Events
5. In order to collect the Events we are iterating through the ExecutionOutcome's logs trying to parse Event using regular expression. We're returningundefined
6. if we fail to parseEventLogData
7. Finally onceevents
8. list is collected we're filtering it dropping theundefined

Fine, so now we have only a list of our objects that contain some Receipt data and the list of successfully parsedEventLogData .

The goal for our indexer is to return the useful data about a minted NFT that follows NEP-171 standard. We need to drop irrelevant standard Events:

. filter (relevantOutcome => relevantOutcome . events . some (event => event . standard

===

"nep171"

&& event . event

===

"nft_mint"))

Almost done

So far we have collected everything we need corresponding to our requirements.

We can print everything in the end of thehandleStreamerMessage :

relevantOutcomes . length

&&

console . dir (relevantOutcomes ,

{ depth :

10

}) The final look of thehandleStreamerMessage function:

async

function

handleStreamerMessage (streamerMessage : types . StreamerMessage) :

Promise < void

{ const relevantOutcomes = streamerMessage . shards . flatMap (shard => shard . receiptExecutionOutcomes) . map (outcome =>

({ receipt :

{ id : outcome . receipt . receiptId , receiverId : outcome . receipt . receiverId , } , events : outcome . executionOutcome . outcome . logs . map ((log :

string) :

EventLogData

=>

{ const

[_ , probablyEvent]

= log . match (/ ^ EVENT_JSON: (. *)

/)

??

[ ] try

{ return

JSON . parse (probablyEvent) }

catch

(e)

{ return } }) . filter (event => event !==

undefined) })) . filter (relevantOutcome => relevantOutcome . events . some (event => event . standard

===

"nep171"

&& event . event

===

"nft_mint"))

relevantOutcomes . length

&&

console . dir (relevantOutcomes ,

{ depth :

10

}) } And if we run our indexer we will be catchingnft_mint event and print the data in the terminal.

npm run start note Having troubles running the indexer? Please, check you haven't skipped the Credentials part :) Not so fast! Remember we were talking about having the links to the marketplaces to see the minted tokens? We're gonna extend our data with links whenever possible. At least we're gonna show you how to deal with the NFTs minted on Paras and Mintbase .

Crafting links to Paras and Mintbase for NFTs minted there

At this moment we have an array of objects we've crafted on the fly that exposes receipt, execution status and event logs. We definitely know that all the data we have at this moment are relevant for us, and we want to extend it with the links to that minted NFTs at least for those marketplaces we know.

We know and love Paras and Mintbase.

Paras token URL

We did the research for you and here's how the URL to token on Paras is crafting:

https://paras.id/token/[1]::[2]/[3] Where:

- [1] - Paras contract address (x.paras.near
-)
- [2] - First part of thetoken_id
- (EventLogData.data
- for Paras is an array of objects withtoken_ids
- key in it. Those IDs represented by numbers with column:
- between them)
- [3] -token_id
- itself

Example:

https://paras.id/token/x.paras.near::387427/387427:373 Let's add the interface for later use somewhere afterinterface EventLogData :

interface

ParasEventLogData

{ owner_id :

string , token_ids :

string [ ] , } ;

Mintbase token URL

And again we did the research for you:

https://www.mintbase.io/thing/[1]:[2] Where:

- [1] -meta_id
- (EventLogData.data
- for Mintbase is an array of stringified JSON that containsmeta_id
-)
- [2] - Store contract account address (basically Receipt's receiver ID)

Example:

https://www.mintbase.io/thing/70eES-icwSw9iPIkUluMHOV055pKTTgQgTiXtwy3Xus:vnartistsdao.mintbase1.near Let's add

the interface for later use somewhere afterinterface EventLogData :

interface

MintbaseEventLogData

{ owner_id :

string , memo :

string , } Now it's a perfect time to add another.map() , but it might be too much. So let's proceed with a forloop to craft the output data we want to print.

let output =

[ ] for

(let relevantOutcome of relevantOutcomes)

{ let marketplace =

"Unknown" let nfts =

[ ]

} We're going to print the marketplace name, Receipt id so you would be able to search for it onNEAR Explorer and the list of links to the NFTs along with the owner account name.

Let's start crafting the links. Time to say "Hi!" toRiqi (just because we can):

let output =

[ ] for

(let relevantOutcome of relevantOutcomes)

{ let marketplace =

"Unknown" let nfts =

[ ] if

(relevantOutcome . receipt . receiverId . endsWith (".paras.near"))

{ marketplace =

"Paras" nfts = relevantOutcome . events . flatMap (event =>

{ return

(event . data

as

ParasEventLogData [ ]) . map (eventData =>

({ owner : eventData . owner_id , links : eventData . token_ids . map (tokenId =>

https://paras.id/token/ { relevantOutcome . receipt . receiverId } :: { tokenId . split (":") [0] } / { tokenId }) })) }) } A few words about what is going on here. If the Receipt's receiver account name ends with.paras.near (e.g.x.paras.near) we assume it's from Paras marketplace, so we are changing the corresponding variable.

After that we iterate over the Events and itsdata using theParasEventLogData we've defined earlier. Collecting a list of objects with the NFTs owner and NFTs links.

Mintbase turn, we hopeNate and his team havemigrated to NEAR Lake Framework already, saying "Hi!" and crafting the link:

}

else

if

(relevantOutcome . receipt . receiverId . match (/ . mintbase \d + . near

/))

{ marketplace =

"Mintbase" nfts = relevantOutcome . events . flatMap (event =>

{ return

(event . data

as

MintbaseEventLogData [ ]) . map (eventData =>

{ const memo =

JSON . parse (eventData . memo) return

{ owner : eventData . owner_id , links :

[https://mintbase.io/thing/ { memo ["meta_id"] } : { relevantOutcome . receipt . receiverId }] } }) }) } Almost the same story as with Paras, but a little bit more complex. The nature of Mintbase marketplace is that it's not a single marketplace! Every Mintbase user has their own store and a separate contract. And it looks like those contract addresses follow the same principle they end with.mintbaseN.near whereN is number (e.g.nate.mintbase1.near).

After we have defined that the ExecutionOutcome receiver is from Mintbase we are doing the same stuff as with Paras:

1. Changing themarketplace
2. variable
3. Collecting the list of NFTs with owner and crafted links

And if we can't determine the marketplace, we still want to return something, so let's return Events data as is:

}

else

{ nfts = relevantOutcome . events . flatMap (event => event . data) } It's time to push the collected data to theoutput

output . push ({ receiptId : relevantOutcome . receipt . id , marketplace , nfts , }) And make it print the output to the terminal:

if

(output . length)

{ console . log (We caught freshly minted NFTs!) console . dir (output ,

{ depth :

5

}) } All together:

let output =

[ ] for

(let relevantOutcome of relevantOutcomes)

{ let marketplace =

"Unknown" let nfts =

[ ] if

(relevantOutcome . receipt . receiverId . endsWith (".paras.near"))

{ marketplace =

"Paras" nfts = relevantOutcome . events . flatMap (event =>

{ return

(event . data

as

ParasEventLogData [ ]) . map (eventData =>

({ owner : eventData . owner_id , links : eventData . token_ids . map (tokenId =>

https://paras.id/token/ { relevantOutcome . receipt . receiverId } :: { tokenId . split (":") [0] } / { tokenId }) })) }) }

else

if

(relevantOutcome . receipt . receiverId . match (/ . mintbase \d + . near

/))

{ marketplace =

"Mintbase" nfts = relevantOutcome . events . flatMap (event =>

{ return

(event . data

as

MintbaseEventLogData [ ]) . map (eventData =>

{ const memo =

JSON . parse (eventData . memo) return

{ owner : eventData . owner_id , links :

[https://mintbase.io/thing/ { memo ["meta_id"] } : { relevantOutcome . receipt . receiverId }] } }) }) }

else

{ nfts = relevantOutcome . events . flatMap (event => event . data) } output . push ({ receiptId : relevantOutcome . receipt . id , marketplace , createdOn , nfts , }) } if

(output . length)

{ console . log (We caught freshly minted NFTs!) console . dir (output ,

{ depth :

5

}) } OK, how about the date and time of the NFT mint? Let's add to the beginning of thehandleStreamerMessage function

const createdOn =

new

Date (streamerMessage . block . header . timestamp

/

1000000) Update theoutput.push() expression:

output . push ({ receiptId : relevantOutcome . receipt . id , marketplace , createdOn , nfts , }) And not that's it. Run the indexer to watch for NFT minting and never miss a thing.

npm run start note Having troubles running the indexer? Please, check you haven't skipped the Credentials part :) Example output:

We caught freshly minted NFTs! [{ receiptId: '2y5XzzL1EEAxgq8EW3es2r1dLLkcecC6pDFHR12osCGk', marketplace: 'Paras', createdOn: 2022-05-24T09:35:57.831Z, nfts: [{ owner: 'dccc.near', links: ['https://paras.id/token/x.paras.near::398089/398089:17'] }] }] We caught freshly minted NFTs! [{ receiptId:

'BAVZ92XdbkAPX4DkqW5gjCvrhLX6kGq8nD8HkhQFVt5q', marketplace: 'Mintbase', createdOn: 2022-05-24T09:36:00.411Z, nfts: [{ owner: 'chiming.near', links: ['https://mintbase.io/thing/HOTcn6LTo3qTq8bUbB7VwA1GfSDYx2fYOqvP0L_N5Es:vnartistsdao.mintbase1.near'] }] }]

Conclusion

What a ride, yeah? Let's sum up what we have done:

- You've learnt aboutEvents
- Now you understand how to follow for the Events
- Knowing the fact that as a contract developer you can use Events and emit your own events, now you know how to create an indexer that follows those Events
- We've had a closer look at NFT minting process, you can experiment further and find out how to followft_transfer
- Events

The material from this tutorial can be extrapolated for literally any event that follows theEvents format

Not mentioning you have a dedicated indexer to find out about the newest NFTs minted out there and to be the earliest bird to collect them.

Let's go hunt doo, doo, doo

Source code for the tutorial near-examples/near-lake-nft-indexer : source code for this tutorialEdit this page Last updatedonDec 9, 2023 bygagdiez Was this page helpful? Yes No

Previous Python tutorial Next NFT indexer for Python