Building an NFT indexer for Python

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

The Goal

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

- receipt_id
- 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 GitHubfrolvanya/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 Python 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. Let's create our project folder

mkdir lake-nft-indexer && cd lake-nft-indexer touch main.py

Set up NEAR Lake Framework

In themain.py let's importstream function andnear_primitives fromnear-lake-framework :

from near_lake_framework import near_primitives , LakeConfig , streamer Add the main function

async

def

main ( ) : print ("Starting NFT indexer") Add the instantiation ofLakeConfig below:

config

LakeConfig . mainnet ( ) config . start_block_height =

69030747 config . aws_access_key_id = os . getenv ("AWS_ACCESS_KEY_ID") config . aws_secret_key = os . getenv ("AWS_SECRET_ACCESS_KEY") Just a few words on the config, functionmainnet() has sets3_bucket_name ,s3_region_name for mainnet. You can go toNEAR Explorer and getthe most recent block height to setconfig.start_block_height .

Let's callstreamer function with theconfig

stream_handle , streamer_messages_queue = streamer (config) while

True : streamer_message =

await streamer_messages_queue . get ( ) And an actual start of our indexer in the end of themain.py

loop

asyncio . get_event_loop ( ) loop . run_until_complete (main ( )) The finalmain.py at this moment should look like the following:

from near_lake_framework import LakeConfig , streamer , near_primitives async

def

main ( ) : print ("Starting NFT indexer")

config

LakeConfig . mainnet ( ) config . start_block_height =

69030747 config . aws_access_key_id = os . getenv ("AWS_ACCESS_KEY_ID") config . aws_secret_key = os . getenv ("AWS_SECRET_ACCESS_KEY")

stream_handle , streamer_messages_queue = streamer (config) while

True : streamer_message =

await streamer_messages_queue . get ( )

loop

asyncio . get_event_loop ( ) loop . run_until_complete (main ( )) Now we need to create a callback function that we'll be called to handleStreamerMessage our indexer receives.

async

def

handle_streamer_message (streamer_message : near_primitives . StreamerMessage) : pass

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 theData 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 aReceipt
- A result of the Receipt execution isExecutionOutcome
- 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.

Catching only the data we need

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

async

def

handle_streamer_message (streamer_message : near_primitives . StreamerMessage) : for shard in streamer_message . shards : for receipt_execution_outcome in shard . receipt_execution_outcomes : for log in receipt_execution_outcome . execution_outcome . outcome . logs : pass We have iterated through the logs of all ExecutionOutcomes ofShards (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 theEvents docs .

async

def

handle_streamer_message (streamer_message : near_primitives . StreamerMessage) : for shard in streamer_message . shards : for receipt_execution_outcome in shard . receipt_execution_outcomes : for log in receipt_execution_outcome . execution_outcome . outcome . logs : if

not log . startswith ("EVENT_JSON:") : continue try : parsed_log = json . loads (log [len ("EVENT_JSON:")

:]) except json . JSONDecodeError : print (f"Receipt ID{ receipt_execution_outcome . receipt . receipt_id } \nError during parsing logs from JSON string to dict") continue Let us explain what we are doing here:

1. We are walking through the logs in ExecutionOutcomes
2. We are filtering ExecutionOutcomes that contain logs of Events format
3. In order to collect the Events we are iterating through the ExecutionOutcome's logs trying to parse Event usingjson.loads

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:

if

(parsed_log . get ("standard")

!=

"nep171" or parsed_log . get ("event")

!=

"nft_mint") : continue

Almost done

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

The final look of thehandle_streamer_message function:

async

def

handle_streamer_message (streamer_message : near_primitives . StreamerMessage) : for shard in streamer_message . shards : for receipt_execution_outcome in shard . receipt_execution_outcomes : for log in receipt_execution_outcome . execution_outcome . outcome . logs : if

not log . startswith ("EVENT_JSON:") : continue try : parsed_log = json . loads (log [len ("EVENT_JSON:")

:]) except json . JSONDecodeError : print (f"Receipt ID{ receipt_execution_outcome . receipt . receipt_id } \nError during parsing logs from JSON string to dict") continue

if

(parsed_log . get ("standard")

!=

"nep171" or parsed_log . get ("event")

!=

"nft_mint") : continue

print (parsed_log) Now let's callhandle_streamer_message inside the loop inmain function

await handle_streamer_message (streamer_message) And if we run our indexer we will be catchingnft_mint event and print logs in the terminal.

python3 main.py note Having troubles running the indexer? Please, check you haven't skipped theCredentials 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 onParas andMintbase .

Crafting links to Paras and Mintbase for NFTs minted there

At this moment we have an access to logs that follows NEP-171 standard. 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
- (parsed_log["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

Mintbase token URL

And again we did the research for you:

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

- [1] -meta_id
- (parsed_log["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 start crafting the links:

def

format_paras_nfts (data , receipt_execution_outcome) : links =

[ ]

for data_element in data : for token_id in data_element . get ("token_ids" ,

[ ]) : first_part_of_token_id = token_id . split (":") [0] links . append (f"https://paras.id/token/ { receipt_execution_outcome . receipt . receiver_id } :: { first_part_of_token_id } / { token_id } ")

return

{ "owner" : data [0] . get ("owner_id") ,

"links" : links }

def

format_mintbase_nfts (data , receipt_execution_outcome) : links =

[ ] for data_block in data : try : memo = json . loads (data_block . get ("memo")) except json . JSONDecodeError : print (f"Receipt ID: { receipt_execution_outcome . receipt . receipt_id } \nMemo: { memo }\nError during parsing Mintbase memo from JSON string to dict") return

meta_id

memo . get ("meta_id") links . append (f"https://www.mintbase.io/thing/ { meta_id } : { receipt_execution_outcome . receipt . receiver_id } ")

return

{ "owner" : data [0] . get ("owner_id") ,

"links" : links } We're going to print the receipt_id, so you would be able to search for it on NEAR Explorer , marketplace name and the list of links to the NFTs along with the owner account name.

if receipt_execution_outcome . receipt . receiver_id . endswith (".paras.near") : output =

{ "receipt_id" : receipt_execution_outcome . receipt . receipt_id , "marketplace" :

"Paras" , "nfts" : format_paras_nfts (parsed_log ["data"] , receipt_execution_outcome) , } 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.

Mintbase turn, we hope Nate and his team have migrated to NEAR Lake Framework already, saying "Hi!" and crafting the link:

elif re . search (".mintbase\d+.near" , receipt_execution_outcome . receipt . receiver_id) : output =

{ "receipt_id" : receipt_execution_outcome . receipt . receipt_id , "marketplace" :

"Mintbase" , "nfts" : format_mintbase_nfts (parsed_log ["data"] , receipt_execution_outcome) , } else : continue 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. Setting themarketplace
2. variable to Mintbase
3. Collecting the list of NFTs with owner and crafted links

And make it print the output to the terminal:

print (json . dumps (output , indent = 4)) All together:

def

format_paras_nfts (data , receipt_execution_outcome) : links =

[ ]

for data_element in data : for token_id in data_element . get ("token_ids" ,

[ ]) : first_part_of_token_id = token_id . split (":") [0] links . append (f"https://paras.id/token/ { receipt_execution_outcome . receipt . receiver_id } :: { first_part_of_token_id } / { token_id } ")

return

{ "owner" : data [0] . get ("owner_id") ,

"links" : links }

def

format_mintbase_nfts (data , receipt_execution_outcome) : links =

[ ] for data_block in data : try : memo = json . loads (data_block . get ("memo")) except json . JSONDecodeError : print (f"Receipt ID: { receipt_execution_outcome . receipt . receipt_id } \nMemo: { memo }\nError during parsing Mintbase memo from JSON string to dict") return

meta_id

memo . get ("meta_id") links . append (f"https://www.mintbase.io/thing/ { meta_id } : { receipt_execution_outcome . receipt . receiver_id } ")

return

{ "owner" : data [0] . get ("owner_id") ,

"links" : links }

async

def

handle_streamer_message (streamer_message : near_primitives . StreamerMessage) : for shard in streamer_message . shards : for receipt_execution_outcome in shard . receipt_execution_outcomes : for log in receipt_execution_outcome . execution_outcome . outcome . logs : if

not log . startswith ("EVENT_JSON:") : continue try : parsed_log = json . loads (log [len ("EVENT_JSON:")

:]) except json . JSONDecodeError : print (f"Receipt ID{ receipt_execution_outcome . receipt . receipt_id } \nError during parsing logs from JSON string to dict") continue

if

(parsed_log . get ("standard")

!=

"nep171" or parsed_log . get ("event")

!=

"nft_mint") : continue

if receipt_execution_outcome . receipt . receiver_id . endswith (".paras.near") : output =

{ "receipt_id" : receipt_execution_outcome . receipt . receipt_id , "marketplace" :

"Paras" , "nfts" : format_paras_nfts (parsed_log ["data"] , receipt_execution_outcome) , } elif re . search (".mintbase\d+.near" , receipt_execution_outcome . receipt . receiver_id) : output =

{ "receipt_id" : receipt_execution_outcome . receipt . receipt_id , "marketplace" :

"Mintbase" , "nfts" : format_mintbase_nfts (parsed_log ["data"] , receipt_execution_outcome) , } else : continue

print (json . dumps (output , indent = 4)) And not that's it. Run the indexer to watch for NFT minting and never miss a thing.

python3 main.py note Having troubles running the indexer? Please, check you haven't skipped the Credentials part :) Example output:

{ "receipt_id": "8rMK8rxb3WmFcSfM3ahFoeeoBF92pad3tpsqKoSWurr2", "marketplace": "Mintbase", "nfts": { "owner": "vn-artists-dao.near", "links": ["https://www.mintbase.io/thing/aqdCBHB9_2XZY7pwXRRu5rGDeLQl7Q8KgNud1wKgnGo:vnartistsdao.mintbase1.near"] } }
{ "receipt_id": "ArRh94Fe1LKF9yPrAdzrMozWoxMVQbEW2Z2Zf4fsSsce", "marketplace": "Paras", "nfts": { "owner": "eeaeb516e0945893ac01eaf547f499abdbd344831c5fcbefa1a5c0a9f303cc5c", "links": ["https://paras.id/token/x.paras.near::432714/432714:1"] } }

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 tutorial Edit this page Last updatedonMar 29, 2024 byVadim Volodin Was this page helpful? Yes No

Previous NFT Indexer Next Running Lake Indexer