Posts Indexer

info NEAR QueryAPI is currently under development. Users who want to test-drive this solution need to be added to the allowlist before creating or forking QueryAPI indexers.

You can request access throughthis link .

Overview

This indexer creates a new row in a pre-definedposts table created by the user in the GraphQL database for every new post found on the blockchain. This is a simple example that shows how to specify a single table, filter blockchain transaction data for a specific type of transaction, and save the data to the database.

tip This indexer can be found byfollowing this link .

Defining the Database Schema

The first step to creating an indexer is to define the database schema. This is done by editing theschema.sql file in the code editor. The schema for this indexer looks like this:

CREATE

TABLE "posts"

("id"

SERIAL

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , "content"

TEXT

NOT

NULL , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , CONSTRAINT

"posts_pkey"

PRIMARY

KEY

("id")) ; This schema defines a table calledposts with columns:

- id
- : a unique identifier for each row in the table
- account_id
- : the account ID of the user who created the post
- block_height
- : the height of the block in which the post was created
- receipt_id
- : the receipt ID of the transaction that created the post
- content
- : the content of the post
- block_timestamp
- : the timestamp of the block in which the post was created

Defining the Indexing Logic

The next step is to define the indexing logic. This is done by editing theindexingLogic.js file in the code editor. The logic for this indexer can be divided into two parts:

1. Filtering blockchain transactions for a specific type of transaction
2. Saving the data from the filtered transactions to the database

Filtering Blockchain Transactions

The first part of the logic is to filter blockchain transactions for a specific type of transaction. This is done by using thegetBlock function. This function takes in a block and a context and returns a promise. The block is a Near Protocol block, and the context is a set of helper methods to retrieve and commit state. ThegetBlock function is called for every block on the blockchain.

ThegetBlock function for this indexer looks like this:

import

{

Block

}

from

"@near-lake/primitives" ;

async

function

getBlock (block :

Block , context)

{ function

base64decode (encodedValue)

{ let buff =

Buffer . from (encodedValue ,

"base64") ; return

JSON . parse (buff . toString ("utf-8")) ; }

const

SOCIAL_DB

=

"social.near" ;

const nearSocialPosts = block . actions ( ) . filter ((action)

=> action . receiverId

===

SOCIAL_DB) . flatMap ((action)

=> action . operations . map ((operation)

=> operation ["FunctionCall"]) . filter ((operation)

=> operation ?. method_name ===

"set") . map ((functionCallOperation)

=>

({ ... functionCallOperation , args :

base64decode (functionCallOperation . args) , receiptId : action . receiptId , })) . filter ((functionCall)

=>

{ const accountId =

Object . keys (functionCall . args . data) [0] ; return

(Object . keys (functionCall . args . data [accountId]) . includes ("post")

|| Object . keys (functionCall . args . data [accountId]) . includes ("index")) ; })) ;

...

// Further logic for saving nearSocialPosts to the database } This function first defines a helper function calledbase64decode that decodes base64 encoded data. It then defines a constant calledSOCIAL_DB that is the name of the smart contract that stores the posts in NEAR. It then filters the blockchain transactions for a specific type of transaction. This is done by:

1. Filtering the blockchain transactions for transactions where thereceiverId
2. is theSOCIAL_DB
3. database
4. Mapping the operations of the filtered transactions to theFunctionCall
5. operation
6. Filtering theFunctionCall
7. operations for operations where themethod_name
8. isset
9. Mapping the filteredFunctionCall
10. operations to an object that contains theFunctionCall
11. operation, the decodedargs
12. of theFunctionCall
13. operation, and thereceiptId
14. of the transaction
15. Filtering the mapped objects for objects where theargs
16. contain apost
17. orindex
18. key

This function returns an array of objects that contain theFunctionCall operation, the decodedargs of theFunctionCall operation, and thereceiptId of the transaction. This array is callednearSocialPosts .

Saving the Data to the Database

The second part of the logic is to save the data from the filtered transactions to the database. This is done by using thecontext.db.Posts.insert() function. Thecontext.db.Posts.insert() function will be called for every filtered transaction as defined by the.map() function called on the array ofnearSocialPosts .

The function for this indexer looks like this:

...

// Logic for filtering blockchain transactions, defining nearSocialPosts

if

(nearSocialPosts . length

0)

{ const blockHeight = block . blockHeight ; const blockTimestamp =

Number (block . header ( ) . timestampNanosec) ; await

Promise . all (nearSocialPosts . map (async

(postAction)

=>

{ const accountId =

Object . keys (postAction . args . data) [0] ; console . log (ACCOUNT_ID: { accountId }) ;

// create a post if indeed a post if

(postAction . args . data [accountId] . post

&& Object . keys (postAction . args . data [accountId] . post) . includes ("main"))

{ try

{ console . log ("Creating a post...") ; const postData =

{ account_id : accountId , block_height : blockHeight , block_timestamp : blockTimestamp , receipt_id : postAction . receiptId , content : postAction . args . data [accountId] . post . main , } ; await context . db . Posts . insert (postData) ; console . log (Post by { accountId } has been added to the database) ; }

catch

(e)

{ console . error (Error creating a post by { accountId } : { e }) ; } } })) ; }

Querying data from the indexer

The final step is querying the indexer using the public GraphQL API. This can be done by writing a GraphQL query using the GraphiQL tab in the code editor.

For example, here's a query that fetchesposts from thePosts Indexer , ordered byblock_height :

query

MyQuery

{ < user - name

_near_posts_indexer_posts (order_by :

{ block_height :

desc })

{ content block_height account_id } } Once you have defined your query, you can use the GraphiQL Code Exporter to auto-generate a JavaScript or NEAR Widget code snippet. The exporter will create a helper methodfetchGraphQL which will allow you to fetch data from the indexer's GraphQL API. It takes three parameters:

- operationsDoc
- : A string containing the queries you would like to execute.
- operationName
- : The specific query you want to run.
- variables
- : Any variables to pass in that your query supports, such asoffset
- andlimit
- for pagination.

Next, you can call thefetchGraphQL function with the appropriate parameters and process the results.

Here's the complete code snippet for a NEAR component using thePosts Indexer :

```
const
QUERYAPI_ENDPOINT
=
https://near-queryapi.api.pagoda.co/v1/graphql/ ;
State . init ( { data :
[ ] } ) ;
const query =
query MyPostsQuery { <user-name>_near_posts_indexer_posts(order_by: {block_height: desc}) { content block_height account_id } }
function
fetchGraphQL ( operationsDoc , operationName , variables )
{ return
asyncFetch ( QUERYAPI_ENDPOINT , { method :
"POST" , headers :
{
"x-hasura-role" :
<user-name>_near
} , body :
JSON . stringify ( { query : operationsDoc , variables : variables , operationName : operationName , } ) , } ) ; }
fetchGraphQL ( query ,
"MyPostsQuery" ,
{ } ) . then ( ( result )
=>
{ if
( result . status
===
200 )
{ if
( result . body . data )
{ const data = result . body . data . < user - name
        _near_posts_indexer_posts ; State . update ( { data } ) console . log ( data ) ; } } } ) ;
const
renderData
=
( a )
=>
{ return
```

(< div key = { JSON . stringify (a) }

{ JSON . stringify (a) } < / div

) ; } ;

const renderedData = state . data . map (renderData) ; return

({ renderedData }) ; tip To view a more complex example, see this widget which fetches posts with proper pagination: Posts Widget powered By QueryAPI . Edit this page Last updatedonJan 9, 2024 bygagdiez Was this page helpful? Yes No

Previous Base64 params, wrap up Next Hype Indexer