

Migrate from NEAR Lake framework

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 through [this link](#) . In this article you'll learn how to migrate your [NEAR Lake Framework](#) JavaScript indexer to [Near QueryAPI](#) , a fully managed solution to build indexer functions, extract on-chain data, store it in a database, and be able to query it using GraphQL endpoints.

Supported languages Currently QueryAPI only supports JavaScript, so if your indexer code uses TypeScript, Python, or Rust, you'll have to re-write your indexing logic in JS before you can migrate it.

Basic migration

Let's take a [basic JS indexer](#) built with NEAR Lake Framework as an example. This indexer simply logs the Block height and the number of shards for each indexed block, using an [indexer handler](#) `handleStreamerMessage` .

Migrating this basic indexer to QueryAPI is simple. You only need to migrate the code from the `handleStreamerMessage` function:

```
async
function
handleStreamerMessage ( streamerMessage : types . StreamerMessage ) :
Promise < void
{ console . log (Block # { streamerMessage . block . header . height } Shards: { streamerMessage . shards . length } ) ; }
```

Migrating to QueryAPI

1. To start the migration process, [create a new indexer](#)
2. using [QueryAPI](#)
3. . You should see a similar interface like this:
4. Since QueryAPI keeps a compatibility layer with Lake Framework, you don't need to change any references to `streamerMessage`
5. in your indexer function. Just change the function definition to:

```
function
handleStreamerMessage ( streamerMessage )
{ // ... Lake framework indexer code } 1. Next, add your migrated indexer function to the getBlock(...) 2. method, and simply call your function passing block.streamerMessage 3. as parameter:
```

```
async
function
getBlock ( block :
Block )
{ // Add your code here
function
handleStreamerMessage ( streamerMessage )
{ console . log (Block # { streamerMessage . block . header . height } Shards: { streamerMessage . shards . length } ) ; }
```

`handleStreamerMessage (block . streamerMessage) ;` That's all! The basic Lake Framework JS indexer has been migrated to QueryAPI, and you can test it out by using [Debug Mode](#) . If you run the indexer using local debug mode, you should see an output like:

Block #106812523 Shards: 4

Database storage

If you want to take advantage of QueryAPI's database features, you can also store the indexer results in a Postgres DB.

1. First, create the database schema:

```
CREATE
TABLE "basic"
( "block_height"
BIGINT
NOT
NULL , "shards"
INTEGER
NOT
NULL , PRIMARY
KEY
```

("block_height")) ; 1. In your indexer JavaScript code, use the [context.db](#) 2. object to store the results:

```
const basicData =
{ block_height : streamerMessage . block . header . height , shards : streamerMessage . shards . length , } ;
context . db . Basic . insert ( basicData ) ;
```

Advanced migration

For this example, let's take the TypeScript [NFT indexer](#) built with NEAR Lake Framework as reference. This indexer is watching for `mint` [Events](#) and prints some relevant data about minted NFTs.

As with the previous example, moving this NFT indexer to QueryAPI requires to migrate the code from the [handleStreamerMessage](#) function. But since it was done in TypeScript, it also needs some additional work as it needs to be re-written in JavaScript.

Migrating to QueryAPI

To migrate the code, you can take advantage of the [near-lake-primitives](#) provided by QueryAPI, and simplify the indexer logic. For example:

- to get all [Events](#)
- in a Block
- , you can simply call `block.events()`
- .
- you don't need to iterate through shards and execution outcomes, nor manually parse the `EVENT_JSON`
- logs to detect events (it's handled by QueryAPI)

Here's a JavaScript implementation of the NFT indexer using QueryAPI features, that provides the same output as the original indexer:

```
async
function
getBlock ( block :
Block )
{ let output =
[] ;
for
```

```

( let ev of block . events ( ) )

{ const r = block . actionByReceiptId ( ev . relatedReceiptId ) ; const createdOn =

new

Date ( block . streamerMessage . block . header . timestamp

/

1000000 ) ;

try

{ let event = ev . rawEvent ;

if

( event . standard

===

"nep171"

&& event . event

===

"nft_mint" )

{ let nfts =

[ ] ; let marketplace =

"unknown" ;

if

( r . receiverId . endsWith ( ".paras.near" ) ) { marketplace =

"Paras" ; nfts = event . data . map ( eventData

=>

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

=>

https://paras.id/token/ { r . receiverId } :: { tokenId . split ( ":" ) [ 0 ] / { tokenId } } ) } ) ) ; } else

if

( r . receiverId . match ( / . mintbase \d + . near

/ ) ) { marketplace =

"Mintbase" ; nfts = event . data . map ( eventData

=>

{ const memo =

JSON . parse ( eventData . memo ) return

{ owner : eventData . owner_id , links :

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

output . push ( { receiptId : ev . relatedReceiptId , marketplace , createdOn , nfts , } ) ;

} }

catch

```

```
( e )
{ console . log ( e ) ; } }
if
( output . length )
{ console . log ( We caught freshly minted NFTs! ) ; console . dir ( output ,
{
depth :
5
} ) ; } } That's all! The NFT indexer has been migrated to QueryAPI, and you can test it out by using Debug Mode . If you run the indexer using local debug mode, you should see an output like:
```

Block Height #66264722

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/HOTcn6LT03qTq8bUbB7VwA1GfSDYx2fYOqvP0L_N5Es:vnartistsdao.mintbase1.near" ] } ] } ]
```

Database storage

If you want to take advantage of QueryAPI's database features, you can also store the indexer results in a Postgres DB.

1. First, create the database schema:

```
CREATE
TABLE "nfts"
( "id"
SERIAL
NOT
NULL , "marketplace"
TEXT , "block_height"
BIGINT , "timestamp"
DATETIME , "receipt_id"
TEXT , "nft_data"
TEXT , PRIMARY
KEY
( "id" ,
```

"block_height")) ; 1. In your indexer JavaScript code, use the [context.db](#) 2. object to store the results:

```
// ... previous code ... output . push ( { receiptId : ev . relatedReceiptId , marketplace , createdOn , nfts , } ) ;
const nftMintData =
{ marketplace : marketplace , block_height : block . header ( ) . height , timestamp : createdOn , receipt_id : r . receiptId ,
nft_data :
JSON . stringify ( event . data ) , } ;
context . db . Nfts . insert ( nftMintData ) ; } }
catch
```

(e)

{ console . log (e) ; } } // ... code continues ... tip You can find the migrated NFT indexer source code [by clicking here](#) . [Edit this page](#) Last updated on Dec 5, 2023 by Damián Parrino Was this page helpful? Yes No

[Previous Access & Query Historical data](#) [Next Lake Overview](#)