Extending Lake indexer with start options

The End

This tutorial ends with the example code of the simple indexer built on top of NEAR Lake Framework that can start:

- from specified block height (out of the box)./target/release/indexer mainnet from-block 65359506
- from the latest final block from the network./target/release/indexer mainnet from-latest
- from the block indexer has indexed the last before it was interrupted./target/release/indexer mainnet from-interruption

Motivation

To find out whether you need an indexer for you project and to create one means you're covering only one side of things - the development.

There is another important side - the maintenance. This involves:

- indexer needs to be upgraded with a newer version of dependencies
- indexer needs to be updates with a new features you've made
- your server needs some maintenance
- incident had happened
- etc.

Almost in all of the above cases you might want to start or restart your indexer not only from the specific block you need to provide, but from the block it was stopped, or from the latest final block in the network.

NEAR Lake Framework doesn't provide such options. Actually, we didn't empower the library with these options to start indexer intentionally.

Intent We want to keepNEAR Lake Framework crate in the narrowest possible way. The goal for the library is to do a single job and allow it to be empowered with any features but outside of the crate itself Though, the possibility to start indexer from the latest block or from the block after the one it has indexed the last, might be very useful.

Also, duringthe April Data Platform Community Meeting we had a question whether we plan to add this feature to the library. We've promised to create a tutorial showing how to do it by your own. So here it is.

Preparation

In this tutorial we're not going to focus our attention on the indexer itself, but on the start options instead.

note To simplify the code samples in the tutorial, we're writing entire application in a single filesrc/main.rs .

Please, do not take it as a design advice. We do it only for the simplicity Let's prepare a project with a base dependencies, so we can focus on the main goal of this tutorial.

Create a new Rust project

cargo new --bin indexer && cd indexer Replace the content of theCargo.toml file with this:

[package] name = "indexer" version = "0.1.0" edition = "2021" rust-version = "1.60.0"

[dependencies] clap = { version = "3.1.6", features = ["derive"] } futures = "0.3.5" itertools = "0.9.0" tokio = { version = "1.1", features = ["sync", "time", "macros", "rt-multi-thread"] } tokio-stream = { version = "0.1" } tracing = "0.1.13" tracing-subscriber = "0.2.4" serde = { version = "1", features = ["derive"] } serde_json = "1.0.55"

near-lake-framework = "0.3.0" Replace the content ofsrc/main.rs with this:

use

clap :: { Parser ,

Subcommand } ; use

futures :: StreamExt ; use

tracing_subscriber :: EnvFilter ;

// TODO: StartOptions

[tokio::main]

async

fn

main ( )

->

Result < ( ) ,

tokio :: io :: Error

{ init_tracing ( ) ;

let opts =

Opts :: parse ( ) ;

// TODO: Config

let stream =

near_lake_framework :: streamer (config) ;

let

mut handlers =

tokio_stream :: wrappers :: ReceiverStream :: new (stream) . map (handle_streamer_message) . buffer_unordered (1usize) ;

while

let

Some (_handle_message)

= handlers . next ( ) . await

{ }

Ok (( )) }

async

fn

handle_streamer_message (streamer_message :

near_lake_framework :: near_indexer_primitives :: StreamerMessage ,)

{ eprintln! ("{} / shards {}" , streamer_message . block . header . height , streamer_message . shards . len ( )) ; std :: fs :: write ("last_indexed_block" , streamer_message . block . header . height . to_string ( ) . as_bytes ( )) . unwrap ( ) ; }

fn

init_tracing ( )

{ let

mut env_filter =

EnvFilter :: new ("near_lake_framework=info") ;

if

let

Ok (rust_log)

=

std :: env :: var ("RUST_LOG")

{ if

! rust_log . is_empty ( )

{ for directive in rust_log . split (',') . filter_map (| s |

match s . parse ( )

{ Ok (directive)

=>

Some (directive) , Err (err)

=>

{ eprintln! ("Ignoring directive{}: {}" , s , err) ; None } })

{ env_filter = env_filter . add_directive (directive) ; } } }

tracing_subscriber :: fmt :: Subscriber :: builder ( ) . with_env_filter (env_filter) . with_writer (std :: io :: stderr) . init ( ) ; }
This code is not going to build yet. Meanwhile let's have a quick look of what we've copy/pasted for now:

- We have importedclap
- to set up what command line arguments we're going to accept
- Also, we've important necessary stuff likefutures
- andtracing_subscriber
- init_tracing
- in the end of the file is a helper function that subscribes our application to the logs fromnear-lake-framework
- An asynchronousmain
- function with the indexer boilerplate code, but missing theLakeConfig
- creation part we're going to cover in the tutorial.
- You can find a few// TODO: ...
- sections we've marked for you to find places to write the code from this tutorial.

OK, all the preparations are done. Let's move on.

Design theStartOptions

So we want to be able to pass a command that defines the way our indexer should start. In this tutorial we'll be usingclap .

We need a structure that receives the chain id. This will allow us to use command:

./target/release/indexer mainnet ... OR

./target/release/indexer testnet ... Let's replace the// TODO: StartOptions in thesrc/main.rs with:

[derive(Parser, Debug, Clone)]

[clap(version =

"0.1" , author = "Near Inc.hello@nearprotocol.com")] struct

Opts

{

[clap(subcommand)]

pub chain_id :

ChainId , }

[derive(Subcommand, Debug, Clone)]

enum

ChainId

{

[clap(subcommand)]

Mainnet (StartOptions) ,

[clap(subcommand)]

Testnet (StartOptions) , } Now we want to create aStartOptions structure that will allow us to tell our indexer where to start indexing from. The command should look like:

./target/release mainnet from-latest Our variants are:

- from-block N
- , whereN
- is the block height to start from
- from-latest
- to start from latest final block in the network
- from-interruption
- to start from the block indexer was previously interrupted

Let's replace the comment// TODO: StartOptions with the enum:

[derive(Subcommand, Debug, Clone)]

pub (crate)

enum

StartOptions

{ FromBlock

{ height :

u64

} , FromLatest , FromInterruption , } Pretty simple and straightforward, agree?

Creating aLakeConfig

In order to createLakeConfig we're going to use a config builderLakeConfigBuilder . Fotunately, we've imported it already.

Let's instantiate a builder in place of// TODO: Config comment:

let

mut lake_config_builder =

near_lake_framework :: LakeConfigBuilder :: default ( ) ; Notice thatlake_config_builder is defined as mutable.

Now we need to set the chain we are going to index by matchingChainId provided:

let

mut lake_config_builder =

near_lake_framework :: LakeConfigBuilder :: default ( ) ;

match

& opts . chain_id { ChainId :: Mainnet (start_options)

=>

{ lake_config_builder = lake_config_builder . mainnet ( ) ; } ChainId :: Testnet (start_options)

=>

{ lake_config_builder = lake_config_builder . testnet ( ) ; } } As you can see, depending on the variant of theChainId enum we modify thelake_config_builder with one of the shortcutsmainnet() ortestnet() .

The only parameter left to set is the most important for us in this tutorialstart_block_height

Normally, we just pass the block height numberu64 but we're implementing the start options here.

Start options logic

Let's create a separate function that will hold the logic of identification thestart_block_height and call itget_start_block_height .

Just read the code, don't copy, it's not final approach yet

FromBlock { height: u64 }

Let's start from implementationfrom-block N as the simplest one:

async

fn

get_start_block_height (start_options :

& StartOptions)

->

u64

{ match start_options { StartOptions :: FromBlock

{ height }

=> height ,

} } OK, it's simple enough, what's about other match arms forStartOptions :

async

fn

get_start_block_height (start_options :

& StartOptions)

->

u64

{ match start_options { StartOptions :: FromBlock

{ height }

=> height , StartOptions :: FromLatest

=> } } Er, how should we get the latest block from the network? We should query the JSON RPC and get the final block, extract its height and call it a day.

FromLatest

In order to query the JSON RPC from within Rust code we need to usenear-jsonrpc-client-rs crate

You can find abunch of useful examples in the corresponding folder of the project's repository on GitHub.

Add it toCargo.toml in the end:

near-jsonrpc-client = "0.3.0" The code for getting the final block height would look like the following:

use

near_jsonrpc_client :: { methods ,

JsonRpcClient } ; use

near_lake_framework :: near_indexer_primitives :: types :: { BlockReference ,

Finality } ;

async

fn

final_block_height ( )

->

u64

{ let client =

JsonRpcClient :: connect ("https://rpc.mainnet.near.org") ; let request =

methods :: block :: RpcBlockRequest

{ block_reference :

BlockReference :: Finality (Finality :: Final) , } ;

let latest_block = client . call (request) . await . unwrap ( ) ;

latest_block . header . height } Nice and easy. Though, a hardcoded value of"https://rpc.mainnet.near.org" looks not so great. Especially when we want to support both networks.

But we can handle it by passing the JSON RPC URL to theget_start_block_function like this:

async

fn

get_start_block_height (start_options :

& StartOptions , rpc_url :

& str ,)

->

u64

{ ... }

... match

& opts . chain_id { ChainId :: Mainnet (start_options)

=>

{ lake_config_builder = lake_config_builder . mainnet ( ) . start_block_height (get_start_block_height (start_options , "https://rpc.mainnet.near.org" ,) . await) ; } ChainId :: Testnet (start_options)

=>

{ lake_config_builder = lake_config_builder . testnet ( ) . start_block_height (get_start_block_height (start_options , "https://rpc.testnet.near.org" ,) . await) } } Meh. It's ugly and why should we pass it everytime if it is required in only one case from three possible?

Instead we can pass to theget_start_block_height function the entireOpts .

async

fn

get_start_block_height (opts :

& Opts)

->

u64

{ match opts . chain_id { ChainId :: Mainnet (start_options)

=>

{ match start_options { StartOptions :: FromBlock

{ height }

=> height , StartOptions :: FromLatest

=> } } } } At least we have everything we need. Though, it still looks ugly and will definitely involve code duplication.

What we propose instead to is createimpl Opts with a few useful methods to get JSON RPC URL and to getStartOptions instance.

Now you may proceed copying the code safely

Somewhere under theStartOptions definition add the following:

impl

Opts

{ pub

fn

rpc_url (& self)

->

& str

{ match

self . chain_id { ChainId :: Mainnet (_)

=>

"https://rpc.mainnet.near.org" , ChainId :: Testnet (_)

=>

"https://rpc.testnet.near.org" , } }

pub

fn

start_options (& self)

->

& StartOptions

{ match

& self . chain_id { ChainId :: Mainnet (args)

|

ChainId :: Testnet (args)

=> args } } } And now we can create ourget_start_block_height function with the helper function that will query the final blockfinal_block_height (we're going to reuse it, watch for the hands):

async

fn

get_start_block_height (opts :

& Opts)

->

u64

{ match opts . start_options ( )

{ StartOptions :: FromBlock

{ height }

=>

* height , StartOptions :: FromLatest

=>

final_block_height (opts . rpc_url ( )) . await , // a placeholder StartOptions :: FromInterruption

=>

0 , } }

async

fn

final_block_height (rpc_url :

& str)

->

u64

{ let client =

JsonRpcClient :: connect (rpc_url) ; let request =

methods :: block :: RpcBlockRequest

{ block_reference :

BlockReference :: Finality (Finality :: Final) , } ;

let latest_block = client . call (request) . await . unwrap ( ) ;

latest_block . header . height } You may have noticed theFromInterruption and a comment about the placeholder. The reason we've made is to be able to build the application right now to test out thatFromLatest works as expected.

TestingFromLatest

Credentials Please, ensure you've the credentials set up as described on the Credentials page. Otherwise you won't be able to get the code working. Let's try to build and run our code

cargo build --release

./target/release/indexer mainnet from-latest Once the code is built you should see something like that in your terminal:

65364116 / shards 4 65364117 / shards 4 65364118 / shards 4 65364119 / shards 4 65364120 / shards 4 You can stop it by pressingCTRL+C

And now we can move on toFromInterruption

FromInterruption

In order to let an indexer know at what block it was interrupted, the indexer needs to store the block height somewhere. And it should do it in the and of thehandle_message function.

In the boilerplate code you've copy/pasted in the beginning of this tutorial you can notice a line of code:

std :: fs :: write ("last_indexed_block" , streamer_message . block . header . height . to_string ( ) . as_bytes ( )) . unwrap ( ) ; It saves the last indexed block height into a filelast_indexed_block right near the indexer binary.

In the real world indexer you'd probably go with some other storage, depending on the toolset you're using.

But to show you the concept, we've decided to go with the easiest approach by saving it to the file.

Now we need to implement the reading the value from the file.

note If it is a first start of your indexer and you ask it to start from interruption it wouldn't be able to findlast_indexed_block and would just fail.

It's not the behavior we expect. That's why we assume you want it to start from interruption (if possible) or from the latest. Let's finish up ourget_start_block_height

async

fn

get_start_block_height (opts :

& Opts)

->

u64

{ match opts . start_options ( )

{ StartOptions :: FromBlock

{ height }

=>

* height , StartOptions :: FromLatest

=>

final_block_height (opts . rpc_url ( )) . await , // a placeholder StartOptions :: FromInterruption

=>

{ match

& std :: fs :: read ("last_indexed_block")

{ Ok (contents)

=>

{ String :: from_utf8_lossy (contents) . parse ( ) . unwrap ( ) } Err (e)

=>

{ eprintln! ("Cannot read last_indexed_block.\n{}\nStart indexer from latest final" , e) ; latest_block_height (opts . rpc_url ( )

) . await } } } , } } What we are doing here:

- Trying to read the filelast_indexed_block
- If theResult
- isOk
- , we are reading thecontents
- and parsing it
- If theResult
- isErr
- we print a message about the error and calllast_block_height
- to get the final block from the network (the fallback we were talking earlier)

TestingFromInterruption

In order to ensure everything works as expected we will start index from the genesis to store the last indexed block. And then we will start it from interruption to ensure we're not starting from latest.

Let's build and run from genesis.

Genesis Trick To start NEAR Lake Framework based indexer from the genesis block, you need to just specify thestart_block_height as0 . cargo build --release ./target/release/indexer mainnet from-block 0 You will see something like:

9820210 / shards 1 9820214 / shards 1 9820216 / shards 1 9820219 / shards 1 9820221 / shards 1 9820226 / shards 1 9820228 / shards 1 9820230 / shards 1 9820231 / shards 1 9820232 / shards 1 9820233 / shards 1 9820235 / shards 1 9820236 / shards 1 9820237 / shards 1 9820238 / shards 1 Stop it by pressingCTRL+C

Memorize the last block height you see. In our example it is9820238

Restart the indexer from interruption

./target/release/indexer mainnet from-interruption You should see the indexer logs beginning from the block you've memorized.

Perfect! It's all done. Now you can adjust the code you got in the result to your needs and use it in your indexers.

Summary

You've seen the way how you can empower your indexer with the starting options. As you can see there is nothing complex here.

You can find the source code in thenear-examples/lake-indexer-start-options Edit this page Last updatedonNov 17, 2023 byDamian Parrino Was this page helpful? Yes No

Previous Running Lake Indexer Next Credentials