Migrating to NEAR Lake Framework

We encourage everyone who don't have a hard requirement to useNEAR Indexer Framework consider the migration toNEAR Lake Framework .

In this tutorial we'll show you how to migrate the project usingindexer-tx-watcher-example as a showcase.

Source code The source code for the migrated indexer can be found on GitHubhttps://github.com/near-examples/indexer-tx-watcher-example-lake/tree/0.4.0 Diffs We've posted the diffs for the reference in the endof the article, you can scroll down to them if diffs are all you need in order to migrate your indexer

Changing the dependencies

First of all we'll start from the dependencies inCargo.toml

[package] name = "indexer-tx-watcher-example" version = "0.1.0" authors = ["Near Inchello@nearprotocol.com"] edition = "2018"

[dependencies] actix = "=0.11.0-beta.2" actix-rt = "=2.2.0" # remove it once actix is upgraded to 0.11+ base64 = "0.11" clap = "3.0.0-beta.1" openssl-probe = { version = "0.1.2" } serde = { version = "1", features = ["derive"] } serde_json = "1.0.55" tokio = { version = "1.1", features = ["sync"] } tracing = "0.1.13" tracing-subscriber = "0.2.4"

near-indexer = { git = "https://github.com/near/nearcore", rev = "25b000ae4dd9fe784695d07a3f2e99d82a6f10bd" } * Updateedition * to2021 * Dropactix * crates * Dropopenssl-probe * crate * Addfutures * anditertools * Add features totokio * as we will be using tokio runtime * Addtokio-stream * crate * Replacenear-indexer * withnear-lake-framework

So in the end we'll have this after all:

[package] name = "indexer-tx-watcher-example" version = "0.1.0" authors = ["Near Inchello@nearprotocol.com"] edition = "2021"

[dependencies] base64 = "0.11" clap = { version = "3.1.6", features = ["derive"] } futures = "0.3.5" serde = { version = "1", features = ["derive"] } serde_json = "1.0.55" 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"

near-lake-framework = "0.4.0"

Change the clap configs

Currently we have structureOpts that has a subcommand withRun andInit command. SinceNEAR Lake Framework doesn't needdata and config files we don't needInit at all. So we need to combine some structures intoOpts itself.

... /// NEAR Indexer Example /// Watches for stream of blocks from the chain

[derive(Clap, Debug)]

[clap(version =

"0.1" , author = "Near Inc.hello@nearprotocol.com")] pub (crate)

struct

Opts

{ /// Sets a custom config dir. Defaults to ~/.near/

[clap(short, long)]

pub home_dir :

Option < std :: path :: PathBuf

,

[clap(subcommand)]

pub subcmd :

SubCommand , }

[derive(Clap, Debug)]

pub (crate)

enum

SubCommand

{ /// Run NEAR Indexer Example. Start observe the network Run (RunArgs) , /// Initialize necessary configs Init (InitConfigArgs) , }

[derive(Clap, Debug)]

pub (crate)

struct

RunArgs

{ /// account ids to watch for

[clap(long)]

pub accounts :

String , }

[derive(Clap, Debug)]

pub (crate)

struct

InitConfigArgs

{ ... } ... We are going:

- DropInitConfigArgs
- completely
- Move the content fromRunArgs
- toOpts
- and then dropRunArgs
- Drophome_dir
- fromOpts
- Addblock_height
- toOpts
- to know from which block height to start indexing
- RefactorSubCommand
- to have to variants: mainnet and testnet to define what chain to index
- And addClone
- detive to the structs for later

/// NEAR Indexer Example /// Watches for stream of blocks from the chain

[derive(Clap, Debug, Clone)]

[clap(version =

"0.1" , author = "Near Inc.hello@nearprotocol.com")] pub (crate)

struct

Opts

{ /// block height to start indexing from

[clap(long)]

pub block_height :

u64 , /// account ids to watch for

[clap(long)]

pub accounts :

String ,

[clap(subcommand)]

pub subcmd :

SubCommand , }

[derive(Clap, Debug, Clone)]

pub (crate)

enum

SubCommand

{ Mainnet , Testnet , } In the end of the file we have one implementation we need to replace.

... impl

From < InitConfigArgs

for

near_indexer :: InitConfigArgs

{ ... } We want to be able to castOpts tonear_lake_framework::LakeConfig . So we're going to create a new implementation.

impl

From < Opts

for

near_lake_framework :: LakeConfig

{ fn

from (opts :

Opts)

->

Self

{ let

mut lake_config = near_lake_framework :: LakeConfigBuilder :: default ( ) . start_block_height (opts . block_height) ;

match

& opts . subcmd { SubCommand :: Mainnet

=>

{ lake_config = lake_config . mainnet ( ) ; } SubCommand :: Testnet

=>

{ lake_config = lake_config . testnet ( ) ; } } ;

lake_config . build ( ) . expect ("Failed to build LakeConfig") } } And the final move is to changeinit_logging function to remove redundant log subscriptions:

... pub (crate)

fn

init_logging ( )

{ let env_filter =

EnvFilter :: new ("tokio_reactor=info,near=info,stats=info,telemetry=info,indexer_example=info,indexer=info,near-performance-metrics=info" ,) ; tracing_subscriber :: fmt :: Subscriber :: builder ( ) . with_env_filter (env_filter) . with_writer (std :: io :: stderr) . init ( ) ; } ... Replace it with

... pub (crate)

fn

init_logging ( )

{ let env_filter =

EnvFilter :: new ("near_lake_framework=info") ; tracing_subscriber :: fmt :: Subscriber :: builder ( ) . with_env_filter (env_filter) . with_writer (std :: io :: stderr) . init ( ) ; } ... Finally we're done withsrc/config.rs and now we can move on tosrc/main.rs

Replacing the indexer instantiation

Since we can usetokio runtime and make ourmain function asynchronous it's shorted to show the recreating of themain function than the process of refactoring.

Let's start from import section

Imports before

use

std :: str :: FromStr ;

use

std :: collections :: { HashMap ,

HashSet } ;

use

clap :: Clap ; use

tokio :: sync :: mpsc ; use

tracing :: info ;

use

configs :: { init_logging ,

Opts ,

SubCommand } ;

mod

configs ;

Imports after

We're addingnear_lake_framework imports and remove redundant import fromconfigs .

use

std :: str :: FromStr ;

use

std :: collections :: { HashMap ,

HashSet } ;

use

clap :: Clap ; use

tokio :: sync :: mpsc ; use

tracing :: info ;

use

near_lake_framework :: near_indexer_primitives ; use

near_lake_framework :: LakeConfig ;

use

configs :: { init_logging ,

Opts } ;

Creatingmain()

Let's create an asyncmain() function, callinit_logging and read theOpts .

[tokio::main]

async

fn

main ( )

->

Result < ( ) ,

tokio :: io :: Error

{ init_logging ( ) ;

let opts :

Opts

=

Opts :: parse ( ) ; Let's castLakeConfig fromOpts and instantiateNEAR Lake Framework 'sstream

[tokio::main]

async

fn

main ( )

->

Result < ( ) ,

tokio :: io :: Error

{ init_logging ( ) ;

let opts :

Opts

=

Opts :: parse ( ) ;

let config :

LakeConfig

= opts . clone ( ) . into ( ) ;

let

(_ , stream)

=

near_lake_framework :: streamer (config) ; Copy/paste the code of readingaccounts arg toVec<AccountId

from the oldmain()

[tokio::main]

async

fn

main ( )

->

Result < ( ) ,

tokio :: io :: Error

{ init_logging ( ) ;

let opts :

Opts

=

Opts :: parse ( ) ;

let config :

LakeConfig

= opts . clone ( ) . into ( ) ;

let

(_ , stream)

=

near_lake_framework :: streamer (config) ;

let watching_list = opts . accounts . split (',') . map (| elem |

{ near_indexer_primitives :: types :: AccountId :: from_str (elem) . expect ("AccountId is invalid") }) . collect ( ) ; Now we can calllisten_blocks function we have used before in our indexer while it was built on top ofNEAR Indexer Framework . And returnOk(()) so ourmain() would be happy.

Final async main with NEAR Lake Framework stream

[tokio::main]

async

fn

main ( )

->

Result < ( ) ,

tokio :: io :: Error

{ init_logging ( ) ;

let opts :

Opts

=

Opts :: parse ( ) ;

let config :

LakeConfig

= opts . clone ( ) . into ( ) ;

let

(_ , stream)

=

near_lake_framework :: streamer (config) ;

let watching_list = opts . accounts . split (',') . map (| elem |

{ near_indexer_primitives :: types :: AccountId :: from_str (elem) . expect ("AccountId is invalid") }) . collect ( ) ;

listen_blocks (stream , watching_list) . await ;

Ok (( )) } We're done. That's pretty much entiremain() function. Drop the old one if you haven't yet.

Changes in other function related to data types

Along withNEAR Lake Framework release we have extracted the structures created for indexers into a separate crate. This was done in order to avoid dependency onnearcore as now you can depend on a separate crate that is alreadypublished on crates.io or on NEAR Lake Framework that exposes that crate.

listen_blocks

A function signature needs to be changed to point to new place for data types

async

fn

listen_blocks (mut stream :

mpsc :: Receiver < near_indexer :: StreamerMessage

, watching_list :

Vec < near_indexer :: near_primitives :: types :: AccountId

,)

{ async

fn

listen_blocks (mut stream :

mpsc :: Receiver < near_indexer_primitives :: StreamerMessage

, watching_list :

Vec < near_indexer_primitives :: types :: AccountId

,)

{ And another 3 places wherenear_indexer::near_primitives needs to be replaced withnear_indexer_primitives

if

let

near_indexer_primitives :: views :: ReceiptEnumView :: Action

{ if

let

near_indexer_primitives :: views :: ReceiptEnumView :: Action

{ if

let

near_indexer_primitives :: views :: ActionView :: FunctionCall

{

is_tx_receiver_watched()

And final change for data types in the functionis_tx_receiver_watched()

fn

is_tx_receiver_watched (tx :

& near_indexer_primitives :: IndexerTransactionWithOutcome , watching_list :

& [near_indexer_primitives :: types :: AccountId] ,)

->

bool

{ watching_list . contains (& tx . transaction . receiver_id) }

Credentials

Configure the Credentials in order to access the data from NEAR Lake Framework

Conclusion

And now we have a completely migrated to NEAR Lake Framework indexer.

We are posting the complete diffs for the reference

Diffs

--- a/Cargo.toml +++ b/Cargo.toml @@ -2,18 +2,18 @@ name = "indexer-tx-watcher-example" version = "0.1.0" authors = ["Near Inc hello@nearprotocol.com"] -edition = "2018" +edition = "2021"

[dependencies] -actix = "=0.11.0-beta.2" -actix-rt = "=2.2.0" # remove it once actix is upgraded to 0.11+ base64 = "0.11" -clap = "3.0.0-beta.1" -openssl-probe = { version = "0.1.2" } +clap = { version = "3.1.6", features = ["derive"] } +futures = "0.3.5" serde = { version = "1", features = ["derive"] } serde_json = "1.0.55" -tokio = { version = "1.1", features = ["sync"] } +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"

-near-indexer = { git = "https://github.com/near/nearcore", rev = "25b000ae4dd9fe784695d07a3f2e99d82a6f10bd" } +near-lake-framework = "0.4.0" --- a/src/configs.rs +++ b/src/configs.rs @@ -1,99 +1,50 @@ -use clap::Clap; +use clap::Parser;

use tracing_subscriber::EnvFilter;

/// NEAR Indexer Example /// Watches for stream of blocks from the chain -#[derive(Clap, Debug)] +#[derive(Parser, Debug, Clone)]

[clap(version = "0.1", author = "Near Inc. hello@nearprotocol.com")]

pub(crate) struct Opts { - /// Sets a custom config dir. Defaults to ~/.near/ - #[clap(short, long)] - pub home_dir: Option, - #[clap(subcommand)] - pub subcmd: SubCommand, -} - -#[derive(Clap, Debug)] -pub(crate) enum SubCommand { - /// Run NEAR Indexer Example. Start observe the network - Run(RunArgs), - /// Initialize necessary configs - Init(InitConfigArgs), -} - -#[derive(Clap, Debug)] -pub(crate) struct RunArgs { + /// block height to start indexing from + #[clap(long)] + pub block_height: u64, /// account ids to watch for

[clap(long)]

pub accounts: String, + #[clap(subcommand)] + pub subcmd: SubCommand, }

-#[derive(Clap, Debug)] -pub(crate) struct InitConfigArgs { - /// chain/network id (localnet, testnet, devnet, betanet) - #[clap(short, long)] - pub chain_id: Option, - /// Account ID for the validator key - #[clap(long)] - pub account_id: Option, - /// Specify private key generated from seed (TESTING ONLY) - #[clap(long)] - pub test_seed: Option, - /// Number of shards to initialize the chain with - #[clap(short, long, default_value = "1")] - pub num_shards: u64, - /// Makes block production fast (TESTING ONLY) - #[clap(short, long)] - pub fast: bool, - /// Genesis file to use when initialize testnet (including downloading) - #[clap(short, long)] - pub genesis: Option, - /// Download the verified NEAR genesis file automatically. - #[clap(long)] - pub download_genesis: bool, - /// Specify a custom download URL for the genesis file. - #[clap(long)] - pub download_genesis_url: Option, - /// Download the verified NEAR config file automtically. - #[clap(long)] - pub download_config: bool, - /// Specify a custom download URL for the config file. - #[clap(long)] - pub download_config_url: Option, - /// Specify the boot nodes to bootstrap the network - #[clap(long)] - pub boot_nodes: Option, - /// Specify a custom max_gas_burnt_view limit. - #[clap(long)] - pub max_gas_burnt_view: Option, +#[derive(Parser, Debug, Clone)] +#[pub(crate) enum SubCommand { + Mainnet, + Testnet, }

pub(crate) fn init_logging() { - let env_filter = EnvFilter::new(- "tokio_reactor=info,near=info,stats=info,telemetry=info,indexer_example=info,indexer=info,near-performance-metrics=info", -); + let env_filter = EnvFilter::new("near_lake_framework=info"); tracing_subscriber::fmt::Subscriber::builder() .with_env_filter(env_filter) .with_writer(std::io::stderr) .init(); }

-impl From for near_indexer::InitConfigArgs { - fn from(config_args: InitConfigArgs) -> Self { - Self { - chain_id: config_args.chain_id, - account_id: config_args.account_id, - test_seed: config_args.test_seed, - num_shards: config_args.num_shards, - fast: config_args.fast, - genesis: config_args.genesis, - download_genesis: config_args.download_genesis, - download_genesis_url: config_args.download_genesis_url, - download_config: config_args.download_config, - download_config_url: config_args.download_config_url, - boot_nodes:

config_args.boot_nodes, - max_gas_burnt_view: config_args.max_gas_burnt_view, - } +impl From for near_lake_framework::LakeConfig { + fn from(opts: Opts) -> Self { + let mut lake_config = + near_lake_framework::LakeConfigBuilder::default().start_block_height(opts.block_height); + + match &opts.subcmd { + SubCommand::Mainnet => { + lake_config = lake_config.mainnet(); + } + SubCommand::Testnet => { + lake_config = lake_config.testnet(); + } + }; + + lake_config.build().expect("Failed to build LakeConfig") } } --- a/src/main.rs +++ b/src/main.rs @@ -2,11 +2,14 @@

use std::collections::{HashMap, HashSet};

-use clap::Clap; +use clap::Parser; use tokio::sync::mpsc; use tracing::info;

-use configs::{init_logging, Opts, SubCommand}; +use near_lake_framework::near_indexer_primitives; +use near_lake_framework::LakeConfig; + +use configs::{init_logging, Opts};

mod configs;

@@ -15,60 +18,34 @@ /// We want to catch all*successful* transactions sent to one of the accounts from the list. /// In the demo we'll just look for them and log them but it might and probably should be extended based on your needs.

-fn main() { - // We use it to automatically search the for root certificates to perform HTTPS calls - // (sending telemetry and downloading genesis) - openssl_probe::init_ssl_cert_env_vars(); +#[tokio::main] +async fn main() -> Result<(), tokio::io::Error> { init_logging();

let opts: Opts = Opts::parse();

- let home_dir = opts.home_dir.unwrap_or_else(near_indexer::get_default_home);
- let config: LakeConfig = opts.clone().into();
- match opts.subcmd {
- SubCommand::Run(args) => {
- // Create the Vec of AccountId from the provided--accounts to pass it to listen_blocks
- let watching_list = args
- .accounts
- .split(',')
- .map(|elem| {
- near_indexer::near_primitives::types::AccountId::from_str(elem)
- .expect("AccountId is invalid")
- })

- .collect();

- // Inform about indexer is being started and what accounts we're watching for
- eprintln!(
- "Starting indexer transaction watcher for accounts: \n {:#?}",
- &args.accounts

-);

- // Instantiate IndexerConfig with hardcoded parameters
- let indexer_config = near_indexer::IndexerConfig {
- home_dir,
- sync_mode: near_indexer::SyncModeEnum::FromInterruption,
- await_for_node_synced: near_indexer::AwaitForNodeSyncedEnum::WaitForFullSync,
- };

- let (_, stream) = near_lake_framework::streamer(config);

- // Boilerplate code to start the indexer itself

- let sys = actix::System::new();
- sys.block_on(async move {
- eprintln!("Actix");
- let indexer = near_indexer::Indexer::new(indexer_config);
- let stream = indexer.streamer();
- actix::spawn(listen_blocks(stream, watching_list));
- });
- sys.run().unwrap();

- }
- SubCommand::Init(config) => near_indexer::indexer_init_configs(&home_dir, config.into()),
- }
- let watching_list = opts
- .accounts
- .split(',')
- .map(|elem| {
- near_indexer_primitives::types::AccountId::from_str(elem).expect("AccountId is invalid")
- })
- .collect(); +
- listen_blocks(stream, watching_list).await; +
- Ok(()) }

/// The main listener function the will be reading the stream of blocksStreamerMessage /// and perform necessary checks async fn listen_blocks(- mut stream: mpsc::Receiver, - watching_list: Vec, + mut stream: mpsc::Receiver, + watching_list: Vec,) { eprintln!("listen_blocks"); // This will be a map of correspondence between transactions and receipts @@ -120,7 +97,7 @@ &execution_outcome.receipt.receiver_id, execution_outcome.execution_outcome.outcome.status); - if let near_indexer::near_primitives::views::ReceiptEnumView::Action { + if let near_indexer_primitives::views::ReceiptEnumView::Action { signer_id, .. } = &execution_outcome.receipt.receipt @@ -128,19 +105,20 @@ eprintln!("{}", signer_id); }

- if let near_indexer::near_primitives::views::ReceiptEnumView::Action {
- actions,
- ..
- if let near_indexer_primitives::views::ReceiptEnumView::Action {
- actions, .. } = execution_outcome.receipt.receipt { for action in actions.iter() {
- if let near_indexer::near_primitives::views::ActionView::FunctionCall {
- if let near_indexer_primitives::views::ActionView::FunctionCall { args, .. } = action { if let Ok(decoded_args) = base64::decode(args) {
- if let Ok(args_json) = serde_json::from_slice::(&decoded_args) {
- if let Ok(args_json) =
- serde_json::from_slice::(&decoded_args)
- { eprintln!("{:#?}", args_json); } } @@ -156,8 +134,8 @@ }

fn is_tx_receiver_watched(- tx: &near_indexer::IndexerTransactionWithOutcome, - watching_list: &[near_indexer::near_primitives::types::AccountId], + tx: &near_indexer_primitives::IndexerTransactionWithOutcome, + watching_list: &[near_indexer_primitives::types::AccountId],) -> bool { watching_list.contains(&tx.transaction.receiver_id) } Edit this page Last updatedonNov 17, 2023 byDamian Parrino Was this page helpful? Yes No

Previous Getting started Next Lake Primitive Types