Setting up an SDK client

Once the sdk has been imported into the project, it can be used to interact with the Namada blockchain. Let's assume we have a node running on ip and port127.0.0.1:26657 and we want to send a transaction to the network.

The SDK may be for various purposes, but in this example we will use it to send a transaction to the network.

First, we will need to implement theClient so that we can communicate with a running node.

use reqwest :: { Client , Response

as

ClientResponse };

pub

struct

SdkClient { url :

String , client :

Client , }

impl

SdkClient { pub

fn

new (url :

String) -> Self { Self { client :

Client :: new (), url, } }

pub

async

fn

post (& self, body :

String) ->

Result < ClientResponse , reqwest :: Error

{ self . client . post (format! ("http://{}" , & self . url)) . body (body) . send () .await } } This allows us to useClient fromreqwest (an external library) to send a transaction to the network.

We will need to also define some functions that the client will use to interact with the network.

[async_trait

:: async_trait] impl

ClientTrait

for

SdkClient { type

Error

=

Error ;

async

fn

request (& self, path :

String , data :

Option < Vec < u8

, height :

Option < BlockHeight

, prove :

bool ,) ->

Result < EncodedResponseQuery , Self :: Error

{ let data = data . unwrap_or_default (); // default to empty vec let height = height . map (| height | { tendermint :: block :: Height :: try_from (height . 0) . map_err (| _err |

Error :: InvalidHeight (height)) }) . transpose () ? ; // convert to tendermint::block::Height let response = self . abci_query (Some (std :: str :: FromStr :: from_str (& path) . unwrap ()), data, height, prove,) .await? ;

match response . code { Code :: Ok

=>

Ok (EncodedResponseQuery { data : response . value, info : response . info, proof : response . proof, }), Code :: Err (code) =>

Err (Error :: Query (response . info, code)), } }

async

fn

perform < R

(& self, request :

R) ->

Result < R :: Response , tm_rpc :: Error

where R : tm_rpc :: SimpleRequest , { let request_body = request . into_json (); let response = self . post (request_body) .await ;

match response { Ok (response) => { let response_json = response . text () .await. unwrap (); R :: Response :: from_string (response_json) } Err (e) => { let error_msg = e . to_string (); Err (tm_rpc :: Error :: server (error_msg)) } } } } This client will allow us to make asynchronous calls to the network and handle the responses.

Instantiating a Namada Implementation object

When constructing transactions using the sdk, we almost alwasy need anamada_impl object.

use namada_sdk :: NamadaImpl ; // This module allows us to access the NamadaImpl struct, which is needed for most transactions

let source_address =

Address :: from_str ("tnam1v4ehgw36xq6ngs3ng5crvdpngg6yvsecx4znjdfegyurgwzzx4pyywfexuuyys69gc6rzdfnryrntx") . unwrap (); let http_client = reqwest :: Client :: new (); let wallet =

Wallet :: from_mnemonic ("your mnemonic here") . unwrap (); let wallet : namada_sdk :: wallet :: Wallet < FsWalletUtils

=

FsWalletUtils :: new (PathBuf :: from ("wallet.toml")); let shielded_ctx =

FsShieldedUtils :: new (Path :: new ("masp/") . to_path_buf ()); let namada_impl =

NamadaImpl :: new (http_client, wallet, shielded_ctx, NullIo) .await . expect ("unable to construct Namada object") .

chain_id (ChainId :: from_str (CHAIN_ID) . unwrap ()); This object will be referenced throughout the documentation asnamada_impl .

Using the SDK Setting up a wallet