CosmWasm + ICQ

This section contains a tutorial for writing smart contracts that utilize Interchain Queries Module.

Overview

We are going to learn how to:

1. Install dependencies and import the libraries.
2. Register different Interchain Queries.
3. Get results from the registered Interchain Queries.
4. Manage the registered Interchain Queries.

Note: this section assumes that you have basic knowledge of CosmWasm and have some experience in writing smart contracts. You can check out CosmWasm[docs](#) and[blog posts](#) for entry-level tutorials. Note: before running any query creation transaction you need to top up your contract address. See[Interchain Queries Overview](#) , "Query creation deposit" section.

The complete example

In the snippets below some details might be omitted. Please check out the complete smart contract[example](#) for a complete implementation.

1. Install dependencies and import the libraries

In order to start using the Neutron ICQ module, you need to install some dependencies. Add the following libraries to your dependencies section:

[dependencies] cosmwasm-std

=

"1.2.5"

Other standard dependencies...

This is a library that simplifies working with ICQ,

contains bindings for the Neutron ICQ module (messages, responses, etc.), some default Interchain Queries and provides

various helper functions.

neutron-sdk

=

"0.5.0" Now you can import the libraries:

use

neutron_sdk :: { bindings :: { msg :: NeutronMsg , query :: { NeutronQuery ,

QueryRegisteredQueryResponse } , types :: { Height ,

KVKey } , } , interchain_queries :: { new_register_balance_query_msg , new_register_transfers_query_msg , queries :: { get_registered_query , query_balance , } , register_queries :: new_register_interchain_query_msg , types :: { QueryType ,

TransactionFilterItem ,

TransactionFilterOp ,

TransactionFilterValue , COSMOS_SDK_TRANSFER_MSG_URL ,

RECIPIENT_FIELD , } , } , sudo :: msg :: SudoMsg , NeutronError ,

NeutronResult , } ;

2. Register an Interchain Query

Neutron allows a smart contract to register multiple interchain queries:

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

[serde(rename_all =

"snake_case")] pub

enum

ExecuteMsg

{ RegisterBalanceQuery

{ connection_id :

String , update_period :

u64 , addr :

String , denom :

String , } , RegisterTransfersQuery

{ connection_id :

String , update_period :

u64 , recipient :

String , min_height :

Option < u64

, } }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

execute (deps :

DepsMut < NeutronQuery

, env :

Env , _ :

MessageInfo , msg :

ExecuteMsg ,)

->

NeutronResult < Response < NeutronMsg

{ match msg { ExecuteMsg :: RegisterBalanceQuery

{ connection_id , addr , denom , update_period , }

=>

register_balance_query (deps , env , connection_id , addr , denom , update_period ,) , ExecuteMsg :: RegisterTransfersQuery

{ connection_id , recipient , update_period , min_height , }

=>

register_transfers_query (deps , env , connection_id , recipient , update_period , min_height ,) , } }

pub

fn

register_balance_query (deps :

DepsMut < NeutronQuery

, env :

Env , connection_id :

String , addr :

String , denom :

String , update_period :

u64 ,)

->

NeutronResult < Response < NeutronMsg

{ let msg =

new_register_balance_query_msg (connection_id , addr , denom , update_period) ? ; // wrap into submessage to save {query_id, query_type} on reply that'll later be used to handle sudo kv callback let submsg =

SubMsg :: reply_on_success (msg ,

BALANCES_REPLY_ID) ;

Ok (Response :: default ( ) . add_submessage (submsg)) }

pub

fn

register_transfers_query (deps :

DepsMut < NeutronQuery

, env :

Env , connection_id :

String , recipient :

String , update_period :

u64 , min_height :

Option < u64

,)

->

NeutronResult < Response < NeutronMsg

{ let msg =

new_register_transfers_query_msg (deps , env , connection_id , recipient , update_period , min_height ,) ? ;

Ok (Response :: new ( ) . add_message (msg)) }

[entry_point]

pub

fn

reply (deps :

DepsMut , _ :

Env , msg :

Reply)

->

StdResult < Response

{ deps . api . debug (format! ("WASMDEBUG: reply msg: {:?}" , msg) . as_str ( )) ; match msg . id {
BALANCES_REPLY_ID

=>

write_balance_query_id_to_reply_id (deps , msg) , _ =>

Err (StdError :: generic_err (format! ("unsupported reply message id {}" , msg . id))) , } }

pub

const

KV_QUERY_ID_TO_CALLBACKS :

Map < u64 ,

QueryKind

=

Map :: new ("kv_query_id_to_callbacks") ;

// contains query kinds that we expect to handle insudo_kv_query_result

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]

pub

enum

QueryKind

{ // Balance query Balance , // You can add your handlers to understand what query to deserialize by query_id in sudo callback }

// save query_id to query_type information in reply, so that we can understand the kind of query we're getting in sudo kv call
fn

write_balance_query_id_to_reply_id (deps :

DepsMut , reply :

Reply)

->

StdResult < Response

{ let resp :

MsgRegisterInterchainQueryResponse

=

serde_json_wasm :: from_slice (reply . result . into_result ( ) . map_err (StdError :: generic_err) ? . data . ok_or_else (| |

StdError :: generic_err ("no result")) ? . as_slice ( ) ,) . map_err (| e |

StdError :: generic_err (format! ("failed to parse response: {:?}" , e))) ? ;

// then in success reply handler we do this KV_QUERY_ID_TO_CALLBACKS . save (deps . storage , resp . id ,

& QueryKind :: Balance) ? ;

Ok (Response :: default ( )) } Note: the ICQ module'sRegisterInterchainQueryMsg messagereturns an identifier of newly registered Interchain Query in response. So in a real world scenario you should implement areply handler in your contract to catch the identifier after the registration, so you'll be able to work with the registered query later. In the snippet above, we create theExecuteMsg enum that contains twoRegister messages for two different queries:

- RegisterBalanceQuery
-
    - a simple KV-query to query a balance of an account on remote chain;
- RegisterTransfersQuery
-
    - a TX-query to query transfers transactions to a some recipient on remote chain.

Note: in a real-world scenario you wouldn't want just anyone to be able to make your contract register interchain query, so it might make sense to add ownership checks And implement simple handlersregister_balance_query andregister_transfers_query for these messages. Each handler uses built-in helpers from Neutron-SDK to create necessary register messages:new_register_balance_query_msg andnew_register_transfers_query_msg :

- new_register_balance_query_msg
-
    - is a KV-query, therefore it creates an Interchain Query with necessary KV-keys to read
- from remote chain and build a fullBalance
- response from KV-values (you can see a full implementation of the helper in theSDK source code
-):

pub

fn

new_register_balance_query_msg (...)

->

NeutronResult < NeutronMsg

{ // convert bech32 encoded address to a bytes representation let converted_addr_bytes =

decode_and_convert (addr . as_str ( )) ? ;

// creates a balance KV-key with necessary prefixes we want to read from the storage on remote chain let balance_key =

create_account_denom_balance_key (converted_addr_bytes , denom) ? ;

let kv_key =

KVKey

{ path :

BANK_STORE_KEY . to_string ( ) , key :

Binary (balance_key) , } ;

... } * new_register_transfers_query_msg * - is a TX-query, therefore it creates an Interchain Query with necessary TX-filter * to receive only required transactions from remote chain (you can see a full implementation of the helper in the SDK source code *):

pub

fn

new_register_transfers_query_msg (...)

->

NeutronResult < NeutronMsg

{ // in this case the function creates filter to receive only transactions with transfer msg in it with a particular recipient let

mut query_data :

Vec < TransactionFilterItem

=

vec! [TransactionFilterItem

{ field :

RECIPIENT_FIELD . to_string ( ) , op :

TransactionFilterOp :: Eq , value :

TransactionFilterValue :: String (recipient) , }] ;

... } Note: Neutron SDK is shipped with a lot of helpers to register different Interchain Queries (you can find a full list here). But if you don't find some particular register query helper in the SDK, you can always implement your own using implementations from SDK as a reference. We encourage you to open pull requests with your query implementations to make Neutron SDK better and better!

3. Get results from the registered Interchain Queries

Get results from KV-queries

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

[serde(rename_all =

"snake_case")] pub

enum

QueryMsg

{ GetRegisteredQuery

{ query_id :

u64

} , Balance

{ query_id :

u64

} , }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps < NeutronQuery

, env :

Env , msg :

QueryMsg)

->

NeutronResult < Binary

{ match msg { QueryMsg :: GetRegisteredQuery

{ query_id }

=>

{ Ok (to_binary (& get_registered_query (deps , query_id) ?) ?) } , QueryMsg :: Balance

{ query_id }

=>

Ok (to_binary (& query_balance (deps , env , query_id) ?) ?) , QueryMsg :: GetTransfersNumber

{ }

=>

query_transfers_number (deps) , } } In the snippet above we create theQueryMsg enum that contains three msgs:GetRegisteredQuery ,Balance ,GetTransfersNumber , and aquery entrypoint which handles the defined query msgs.

- the handler ofGetRegisteredQuery
- usesbuilt-in SDK helper
- get_registered_query
- to get all the information about
- any registered query by its id;
- the handler ofBalance
- is much more interesting. It usesbuilt-in SDK helper
- query_balance
- to query interchain balance:
- the handler ofGetTransfersNumber
- will be below in thesection about tx queries handling
- .

pub

fn

query_balance (deps :

Deps < NeutronQuery

, _env :

Env , registered_query_id :

u64 ,)

->

NeutronResult < BalanceResponse

{ // get info about the query let registered_query =

get_registered_query (deps , registered_query_id) ? ; // check that query type is KV check_query_type (registered_query . registered_query . query_type ,

QueryType :: KV) ? ; // reconstruct a nice Balances structure from raw KV-storage values let balances :

Balances

=

query_kv_result (deps , registered_query_id) ? ;

Ok (BalanceResponse

{ // last_submitted_height tells us when the query result was updated last time (block height) last_submitted_local_height : registered_query . registered_query . last_submitted_result_local_height , balances , }) } The most import function here isquery_kv_result :

/// Reads submitted raw KV values for Interchain Query with**query_id** from the storage and reconstructs the result pub

fn

query_kv_result < T :

KVReconstruct

(deps :

Deps < NeutronQuery

, query_id :

u64 ,)

->

NeutronResult < T

{ let registered_query_result =

get_interchain_query_result (deps , query_id) ? ;

KVReconstruct :: reconstruct (& registered_query_result . result . kv_results) } It is built-in into SDK, and it usesKVReconstruct trait to reconstruct KV-storage values into a nice structure. Meaning any structure that implementsKVReconstruct trait can be used withquery_kv_result helper. In our case we want to reconstructBalances from KV-values.Balances is a build-in SDK structure and it alreadyimplements KVReconstruct trait, so no additional functionality is required from developers, you can just import and use it as it is:

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

/// A structure that can be reconstructed from **StorageValues**'s for the **Balance Interchain Query**. /// Contains coins that are held by some account on remote chain. pub

struct

Balances

{ pub coins :

Vec < Coin

, }

impl

KVReconstruct

for

Balances

{ fn

reconstruct (storage_values :

& [StorageValue])

->

NeutronResult < Balances

{ let

mut coins :

Vec < Coin

=

Vec :: with_capacity (storage_values . len ( )) ;

for kv in storage_values { let balance :

CosmosCoin

=

CosmosCoin :: decode (kv . value . as_slice ( )) ? ; let amount =

Uint128 :: from_str (balance . amount . as_str ( )) ? ; coins . push (Coin :: new (amount . u128 ( ) , balance . denom)) ; }

Ok (Balances

{ coins }) } } Note: Neutron SDK is shipped with a lot of query structures to reconstruct different Interchain Queries (you can find a full listhere by looking for structs implementing the KVReconstruct trait). But if you don't find some particular structure in the SDK, you can always implement your own using implementations from SDK as a reference. All you need to do is just implement theKVReconstruct trait for your structure, and after that you can easily use this withquery_kv_result helper like this:let response: YourStructure = query_kv_result(deps, query_id)? Sometimes you might want to get KV Interchain Queries result immediately after it was published by theICQ relayer . That's why we've implementedKV Queries Callbacks , which allows you to get a callback in your contract with the query result when the relayer submits it. KV callbacks are implemented viaSudo calls in your smart-contract:

[entry_point]

pub

fn

sudo (deps :

DepsMut < NeutronQuery

, env :

Env , msg :

SudoMsg)

->

NeutronResult < Response

{ match msg { SudoMsg :: KVQueryResult

{ query_id }

=>

sudo_kv_query_result (deps , env , query_id) , _ =>

Ok (Response :: default ( )) , } }

/// sudo_kv_query_result is the contract's callback for KV query results. Note that only the query /// id is provided, so you need to read the query result from the state. pub

fn

sudo_kv_query_result (deps :

DepsMut < NeutronQuery

, env :

Env , query_id :

u64 ,)

->

NeutronResult < Response

{ deps . api . debug (format! ("WASMDEBUG: sudo_kv_query_result received; query_id: {:?}" , query_id ,) . as_str ( ) ,) ;

// store last KV callback update time KV_CALLBACK_STATS . save (deps . storage , query_id ,

& env . block . height) ? ;

let query_kind =

KV_QUERY_ID_TO_CALLBACKS . may_load (deps . storage , query_id) ? ; match query_kind { Some (QueryKind :: Balance)

=>

{ let balances :

Balances

=

query_kv_result (deps . as_ref ( ) , query_id) ? ; let balances_str = balances . coins . iter ( ) . map (| c | c . amount . to_string ( )

+ c . denom . as_str ( )) . collect :: < Vec < String

( ) . join (", ") ; deps . api . debug (format! ("WASMDEBUG: sudo callback; balances: {:?}" , balances_str) . as_str ( )) ; } None

=>

{ deps . api . debug (format! ("WASMDEBUG: sudo callback without query kind assigned; query_id: {:?}" , query_id) . as_str ( ) ,) ; } } Ok (Response :: default ( )) } In the snippet above we implement asudo entrypoint to catch all theKVQueryResult callbacks, and we definesudo_kv_query_result handler to process the callback. In this particular handler we don't anything, but print some debug info to the log. But there could be any logic you want.

Get results from TX-queries

Unlike KV-queries result, TX-queries results are not saved to the module storage by the Neutron ICQ Module (on Cosmos-SDK level). TX-queries are supported only incallback way , so to get result from TX-queries you have to work withsudo callbacks and save results to the storage by yourself if you need:

[entry_point]

pub

fn

sudo (deps :

DepsMut < NeutronQuery

, env :

Env , msg :

SudoMsg)

->

NeutronResult < Response

{ match msg { SudoMsg :: TxQueryResult

{ query_id , height , data , }

=>

sudo_tx_query_result (deps , env , query_id , height , data) } }

/// sudo_check_tx_query_result is an example callback for transaction query results that stores the /// deposits received as a result on the registered query in the contract's state. pub

fn

sudo_tx_query_result (deps :

DepsMut < NeutronQuery

, _env :

Env , query_id :

u64 , _height :

u64 , data :

Binary ,)

->

NeutronResult < Response

{ // Decode the transaction data let tx :

TxRaw

=

TxRaw :: decode (data . as_slice ( )) ? ; let body :

TxBody

=

TxBody :: decode (tx . body_bytes . as_slice ( )) ? ;

// Get the registered query by ID and retrieve the raw query string let registered_query :

QueryRegisteredQueryResponse

= get_registered_query (deps . as_ref ( ) , query_id) ? ; let transactions_filter = registered_query . registered_query . transactions_filter ;

[allow(clippy::match_single_binding)]

// Depending of the query type, check the transaction data to see whether is satisfies // the original query. If you don't write specific checks for a transaction query type, // all submitted results will be treated as valid. match registered_query . registered_query . query_type { _ =>

{ // For transfer queries, query data looks like[{"field:"transfer.recipient", "op":"eq", "value":"some_address"}] let query_data :

Vec < TransactionFilterItem

= serde_json_wasm :: from_str (transactions_filter . as_str ( )) ? ;

let recipient = query_data . iter ( ) . find (| x | x . field ==

RECIPIENT_FIELD

&& x . op ==

TransactionFilterOp :: Eq) . map (| x |

match

& x . value { TransactionFilterValue :: String (v)

=> v . as_str ( ) , _ =>

"" , }) . unwrap_or ("") ;

let deposits =

recipient_deposits_from_tx_body (body , recipient) ? ; // If we didn't find a Send message with the correct recipient, return an error, and // this query result will be rejected by Neutron: no data will be saved to state. if deposits . is_empty ( )

{ return

Err (NeutronError :: Std (StdError :: generic_err ("failed to find a matching transaction message" ,))) ; }

let

mut stored_transfers :

u64

=

TRANSFERS . load (deps . storage) . unwrap_or_default ( ) ; stored_transfers += deposits . len ( )

as

u64 ; TRANSFERS . save (deps . storage ,

& stored_transfers) ? ;

check_deposits_size (& deposits) ? ; let

mut stored_deposits :

Vec < Transfer

=

RECIPIENT_TXS . load (deps . storage , recipient) . unwrap_or_default ( ) ; stored_deposits . extend (deposits) ; RECIPIENT_TXS . save (deps . storage , recipient ,

& stored_deposits) ? ; Ok (Response :: new ( )) } } } In the snippet above we implement asudo entrypoint to catch all theTXQueryResult callbacks, and we definesudo_tx_query_result handler to process the callback. In the handler we decode the transaction data at first, try to parse messages in the transaction, check that transaction really satisfies our defined filter and do some business logic (in our case we just save transfer to the storage and increase the total incoming transfers number).

IMPORTANT NOTICE: It's necessary to check that the result transaction satisfies your filter. Although Neutron guarantees that transaction is valid (meaning transaction is really included in a block on remote chain, it was executed successfully, signed properly, etc.), Neutroncan not guarantee you that result transaction satisifies defined filter. You must always check this in your contract! Just like with the KV query (the Balance one), TX query results can be retrieved by the contract state. In this respect there is no difference between query types, it's only matter of the way you design your contracts.

/// Returns the number of transfers made on remote chain and queried with ICQ fn

query_transfers_number (deps :

Deps < NeutronQuery

)

->

NeutronResult < Binary

{ let transfers_number =

TRANSFERS . load (deps . storage) . unwrap_or_default ( ) ; Ok (to_binary (& GetTransfersAmountResponse

{ transfers_number }) ?) }

4. Manage registered Interchain Queries

In some cases you may need to update Interchain Queries parameters (update period, KV-keys, tx filter, etc) or even remove a query from the Neutron. Neutron allows you to do these actions viaUpdate andRemove messages:

use

neutron_sdk :: bindings :: msg :: NeutronMsg ;

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

[serde(rename_all =

"snake_case")] pub

enum

ExecuteMsg

{ ... UpdateInterchainQuery

{ query_id :

u64 , new_keys :

Option < Vec < KVKey

, new_update_period :

Option < u64

, } , RemoveInterchainQuery

{ query_id :

u64 , } , ... }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

execute (deps :

DepsMut < NeutronQuery

, env :

Env , _ :

MessageInfo , msg :

ExecuteMsg ,)

->

NeutronResult < Response < NeutronMsg

{ match msg { ... ExecuteMsg :: UpdateInterchainQuery

{ query_id , new_keys , new_update_period , new_recipient , }

=>

update_interchain_query (query_id , new_keys , new_update_period , new_recipinet) , ExecuteMsg :: RemoveInterchainQuery

{ query_id }

=>

remove_interchain_query (query_id) , ... } }

pub

fn

update_interchain_query (query_id :

u64 , new_keys :

Option < Vec < KVKey

, new_update_period :

Option < u64

, new_recipient :

Option < String

,)

->

NeutronResult < Response < NeutronMsg

{ let new_filter = new_recipient . map (| recipient |

{ vec! [TransactionFilterItem

{ field :

RECIPIENT_FIELD . to_string ( ) , op :

TransactionFilterOp :: Eq , value :

TransactionFilterValue :: String (recipient) , }] }) ;

let update_msg =

NeutronMsg :: update_interchain_query (query_id , new_keys , new_update_period , new_filter) ; Ok (Response :: new ( ) . add_message (update_msg)) }

pub

fn

remove_interchain_query (query_id :

u64)

->

NeutronResult < Response < NeutronMsg

{ let remove_msg =

NeutronMsg :: remove_interchain_query (query_id) ; Ok (Response :: new ( ) . add_message (remove_msg)) } In the snippet above we addUpdateInterchainQuery andRemoveInterchainQuery to ourExecuteMsg enum and define corresponding handlersupdate_interchain_query andremove_interchain_query which, in short, just issue properNeutron msgs to update and remove interchain query. In a real world scenario such handlers must have ownership checks.

Learning to make your own queries that are not in Neutron SDK

Same as in the examples above, to make a query, you need to populate KVKey struct:

pub

struct

KVKey

{ /// **path** is a path to the storage (storage prefix) where you want to read value by key (usually name of cosmos-packages module: 'staking', 'bank', etc.) pub path :

String ,

/// **key** is a key you want to read from the storage pub key :

Binary , } ; Let's say we want to make interchain query to wasmd module for contract info. First thing to understand is that you need to know exact version of that module on a chain that you want to query for data. Let's assume we'll query osmosis testnet (osmo-test-5 testnet). Here we discover that chain usesv16.0.0-rc2-testnet version . As we can see this version of osmosis usescustom patched wasmd module .

Now that we have foundthis wasmd module , let's understand how the cosmos-sdk stores data. To simplify: Cosmos SDKstore keeps data as a self-balancing tree where key is an array of bytes. In that tree you can fetch list of elements that share a common prefix and a concrete element if you concatenate prefix with the element key. Usually we'll look intokeeper.go and other files in thekeeper package to see where and what kind of data it keeps in a store. Let's say we want to fetch contract info data. If you look for where contract info is being set, you'll find the store.Sethere , that sets the contract info under the keytypes.GetContractAddressKey(contractAddress) . This function is imported using the keys file and it is a common place for storing all key creation helpers. It's usually placed at/x/modulename/types/keys.go . As we can see the key in store is simplyconcatenation of ContractKeyPrefix ([]byte{0x02}) and address of the contract that you want to query.

Now that we now how to create the key, we can rebuild it's creation using rust in cosmwasm:

// https://github.com/osmosis-labs/wasmd/blob/v0.31.0-osmo-v16/x/wasm/types/keys.go#L28 pub

const

CONTRACT_KEY_PREFIX :

u8

=

0x02 ;

fn

create_contract_address_info_key (addr :

AddressBytes)

->

StdResult < AddressBytes

{ let

mut key :

Vec < u8

=

vec! [CONTRACT_KEY_PREFIX] ; // https://github.com/osmosis-labs/wasmd/blob/v0.31.0-osmo-v16/x/wasm/types/keys.go#L49 key . extend_from_slice (addr . as_slice ( )) ;

Ok (key) } After that we can write use this key in a function that will create message to register this query:

use

neutron_sdk :: interchain_queries :: helpers :: decode_and_convert ; use neutron_sdk :: interchain_queries :: types :: QueryPayload ; use

neutron_sdk :: bindings :: types :: KVKey ; use

neutron_sdk :: bindings :: msg :: NeutronMsg ; use

cosmwasm_std :: Binary ;

// https://github.com/osmosis-labs/wasmd/blob/v0.31.0-osmo-v16/x/wasm/types/keys.go#L13 pub

const

WASM_STORE_KEY :

& str

=

"wasm" ;

pub

fn

new_register_contract_address_info_query_msg (connection_id :

String , addr :

String , update_period :

u64 ,)

->

NeutronResult < NeutronMsg

{ // We need to decode a bech32 encoded string and converts to base64 encoded bytes. // This is needed since addresses are stored this way in Cosmos SDK. let converted_addr_bytes =

decode_and_convert (addr . as_str ( )) ? ;

let balance_key =

create_contract_address_info_key (converted_addr_bytes) ? ;

let kv_key =

KVKey

{ // Path to store, in our case its store of wasmd module (https://github.com/osmosis-labs/wasmd/blob/v0.31.0-osmo-v16/x/wasm/types/keys.go#L13) path :

WASM_STORE_KEY . to_string ( ) , key :

Binary (balance_key) , } ;

// Construct NeutronMsg to register interchain query with our constructed kv_key key, connection_id and update_period NeutronMsg :: register_interchain_query (QueryPayload :: KV (vec! [kv_key]) , connection_id , update_period ,) } By this point we've learned how to register a query with correct key. Now to get some meaningful results, you'll need to understand how to get query results. For that you'll need to implement reconstruction of results usingKVReconstruct trait. This trait has one functionreconstruct that takes raw&[StorageValue] as an input and returnsNeutronResult . Argument into the function will have as many items in it as you'll sent keys when registered the query. In our case it's length will be 1.

These values are stored as a protobuf encoded value. To decode it we'll need to find or describe the type for the protobuf value in rust code.

First find the protobuf type that is used to store the value.ContractInfo is storedhere . There you have two choises:

- Use already available implementations - for osmosis they have osmosis-std lib with our typesee this] (https://github.com/osmosis-labs/osmosis-rust/blob/v0.16.1/packages/osmosis-std/src/types/cosmwasm/wasm/v1.rs#L301))
- ;
- Write your own prost protobuf implementation.

First you'll need to import required libs:

osmosis-std

=

{

version

=

"0.16.1"

} Then you can implement KVReconstruct like this:

use

osmosis_std :: types :: cosmwasm :: wasm :: v1 :: ContractInfo

as

OsmosisContractInfo ; use

neutron_sdk :: interchain_queries :: types :: KVReconstruct ; use

neutron_sdk :: bindings :: types :: StorageValue ; use

neutron_sdk :: { NeutronError ,

NeutronResult } ;

impl

KVReconstruct

for

ContractInfo

{ fn

reconstruct (storage_values :

& [StorageValue])

->

NeutronResult < ContractInfo

{ // our query has one key, that means we expect only one item in the slice if storage_values . len ( )

!=

1

{ return

Err (Std (StdError :: generic_err (format! ("Not one storage value returned for ContractInfo response: {:?}" , storage_values . len ( ))))) ; } // take first key let kv = storage_values . first ( ) . ok_or (Std (StdError :: generic_err (format! ("Not one storage value returned for ContractInfo response: {:?}" , storage_values . len ( ))))) ? ; // decode binary value into protobuf struct let osmosis_res =

OsmosisContractInfo :: decode (kv . value . as_slice ( )) ? ;

// construct result using decoded struct let res =

ContractInfo

{ code_id : osmosis_res . code_id , creator : osmosis_res . creator , admin : osmosis_res . admin , label : osmosis_res . label , created : osmosis_res . created . map (| p |

AbsoluteTxPosition

{ block_height : p . block_height , tx_index : p . tx_index , }) , ibc_port_id : osmosis_res . ibc_port_id , } ;

Ok (res) } }

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]

[serde(rename_all =

"snake_case")] pub

struct

ContractInfo

{ // CodeID is the reference to the stored Wasm code pub code_id :

u64 , // Creator address who initially instantiated the contract pub creator :

String , // Admin is an optional address that can execute migrations pub admin :

String , // Label is optional metadata to be stored with a contract instance. pub label :

String , // Created Tx position when the contract was instantiated. pub created :

Option < AbsoluteTxPosition

, pub ibc_port_id :

String , }

// AbsoluteTxPosition is a unique transaction position that allows for global // ordering of transactions.

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]

[serde(rename_all =

"snake_case")] pub

struct

AbsoluteTxPosition

{ // BlockHeight is the block the contract was created at pub block_height :

u64 , // TxIndex is a monotonic counter within the block (actual transaction index, // or gas consumed) pub tx_index :

u64 , } Now that our ContractInfo implements KVReconstruct, we can try to check that it's working properly. For that we can write something analogous to the[testing.rs test_balance_reconstruct_from_hex]() .

use

base64 :: prelude :: * ; use

base64 :: Engine ;

pub

const

BALANCES_HEX_RESPONSE :

& str

=

"TODO!" ;

// see the code below on how to find this value

[test]

fn

test_balance_reconstruct_from_hex ( )

{ let bytes =

hex :: decode (BALANCES_HEX_RESPONSE) . unwrap ( ) ;

// decode hex string to bytes let base64_input =

BASE64_STANDARD . encode (bytes) ;

// encode bytes to base64 string

let s =

StorageValue

{ storage_prefix :

String :: default ( ) ,

// not used in reconstruct key :

Binary :: default ( ) ,

// not used in reconstruct value :

Binary :: from_base64 (base64_input . as_str ( )) . unwrap ( ) , } ; let bank_balances =

Balances :: reconstruct (& [s]) . unwrap ( ) ; assert_eq! (bank_balances , Balances

{ coins :

vec! [StdCoin

{ denom :

String :: from ("stake") , amount :

Uint128 :: from (99999000u64) , }] }) ; Not that to write a test we need an example of HEX response for our function that we'll use forBALANCES_HEX_RESPONSE constant. To do that you'll need to get value usingRPC_PATH/abci_query GET request with your contructed key and store. data is your KV key in HEX representation of the binary. To construct the key, you can run this code somewhere:

use

neutron_sdk :: interchain_queries :: helpers :: decode_and_convert ;

let addr =

"osmo14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9sq2r9g9" ; let converted_addr_bytes =

decode_and_convert (addr) . unwrap ( ) ; // your newly writtencreate_contract_address_info_key function let actual =

create_contract_address_info_key (converted_addr_bytes) . unwrap ( ) ; println! ("{:?}" ,

hex :: encode (actual)) Then usingabci_query with this key you can write a test namedtest_contract_info_reconstruct() that will use returned value as an input to ContractInfo::reconstruct.

See whole test implementation below:

use

cosmwasm_std :: Binary ; use

crate :: bindings :: types :: StorageValue ; use

neutron_sdk :: interchain_queries :: helpers :: decode_and_convert ; use

neutron_sdk :: interchain_queries :: types :: KVReconstruct ;

const

ABCI_KEY :

& str

=

"02ade4a5f5803a439835c636395a8d648dee57b2fc90d98dc17fa887159b69638b" ; const

ABCI_RESULT :

& str

=

"CAESK29zbW8xcWxtd2prZzd1dTRhd2FqdzVhdW5jdGpkY2U5cTY1N2ozMm54czIiCk9zbW81X1Bhd3MqBAiy9g4=" ;

[test]

fn

test_contract_info_reconstruct ( )

{ let value =

base64 :: decode (ABCI_RESULT) . unwrap ( ) ; let input =

StorageValue

{ storage_prefix :

"wasm" . to_string ( ) , key :

Binary :: from (vec! [ ]) , value :

Binary :: from (value) , } ; let contract_info =

ContractInfo :: reconstruct (& vec! [input]) ; assert! (contract_info . is_ok ( )) ; assert_eq! (contract_info . unwrap ( ) ,

ContractInfo

{ code_id :

1 , creator :

"osmo1qlmwjkg7uu4awajw5aunctjdce9q657j32nxs2" . to_string ( ) , admin :

"" . to_string ( ) , label :

"Osmo5_Paws" . to_string ( ) , created :

Some (AbsoluteTxPosition

{ block_height :

244530 , tx_index :

0

}) , ibc_port_id :

"" . to_string ( ) , }) } Great! Now you can queryContractInfo as simple as this:

use

neutron_sdk :: interchain_queries :: query_kv_result ;

let contract_info :

ContractInfo

=

query_kv_result (deps , query_id) ? ; WARN: please look into correct version of chain when you search on how keys and data model are stored. Otherwise key construction AND/OR data model can change and you'll fail to query data OR reconstruct it! For example, you can see that inv0.45.11-ics sets balance asCoin type and inv0.46.11 it sets only the amount as aString type. So if you don't change the KVReconstruct for this value, it'll break. Previous CosmWasm + ICA Next Integration tests