Part 2: Calling Modules and Contracts

Overview

In the previous part of this tutorial we learned how to implement a simple contract that manages its own state. Real-world applications, however, are rarely that simple; in order to implement something useful, you need to know how to interact withother smart contracts and with Neutronmodules .

Smart contracts

Technically speaking, executing and querying other smart contracts from your own smart contract involves sending messages to thewasmd module, but the interface of this interaction is slightly different from what you will need to do with other modules.

There are 3 things to cover here:

1. Sending messages to smart contracts,
2. Processing responses to those messages,
3. Querying data from smart contracts.

Sending messages to smart contracts, processing the responses and making queries to other contract

In CosmWasm, sending messages from one contract to another is typically done using theWasmMsg::Execute variant within theCosmosMsg . This allows you to execute an action on another contract by sending data (like tokens or structured instructions) to the target contract.

Let's create a new simple contract to interact with the contract from the previous chapter: , which callsIncreaseCount method of the Minimal Contract.

Instantiation of the contract

Since we decided that our contract should interact with the Minimal Contract from the previous part, our contract must store its address somewhere. Let's create a simple config where the address will be stored and we will save the address of the Minimal contract during instantiation:

pub

const

CONFIG :

Item < Config

=

Item :: new ("config") ;

/// Any data that is necessary to set up your new contract should be added /// here.

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

pub

struct

InstantiateMsg

{ // an address of the minimal contract instance pub minimal_contract_address :

Addr , }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

instantiate (deps :

DepsMut , env :

Env , info :

MessageInfo , msg :

InstantiateMsg ,)

->

Result < Response < NeutronMsg

,

ContractError

{ // save minimal contract address to config CONFIG . save (deps . storage , & Config

{ minimal_contract_address : msg . minimal_contract_address , } ,) ? ;

Ok (Response :: new ( ) // We add some attributes to the response with information about the current call. // It's useful for debugging. . add_attribute ("action" ,

"instantiate") . add_attribute ("contract_address" , env . contract . address) . add_attribute ("sender" , info . sender . to_string ( ))) }

Contract execution

Now let's implement the core logic of our contract that calls the Minimal contract instance:

use

cosmwasm_std :: { CosmosMsg ,

WasmMsg , to_json_binary ,

Response ,

DepsMut ,

Env ,

MessageInfo } ; use

cw_storage_plus :: Item ;

pub

fn

send_message_to_contract (deps :

DepsMut , amount :

Uint128)

->

Result < Response ,

ContractError

{ // read the config from, a storage let config =

CONFIG . load (deps . storage) ? ;

// here we compose a message to a minimal contract instance to increase a counter there by specified amount let message

=

CosmosMsg :: Wasm (WasmMsg :: Execute

{ contract_addr : config . minimal_contract_address . into_string ( ) ,

// call the minimal contract by its address msg :

to_json_binary (& MinimalContractExecuteMsg :: IncreaseCount

{ amount }) ? , funds :

vec! [ ] ,

// Optionally, you can send funds along with the message. }) ;

Ok (Response :: new ( ) . add_message (message) . add_attribute ("action" ,

"send_message_to_contract")) } This is it! This simple construction allows to call any method of any contract on Neutron. But what if we also want to handle a response of the call?

In CosmWasm, handling the outcome is straightforward: you generally handle successful execution or errors through the execution result.

Let's modify the code a bit, so we would be able to call a minimal a contract and handle response of the call:

use

cosmwasm_std :: { entry_point , from_json , to_json_binary ,

Addr ,

Binary ,

CosmosMsg ,

DecimalRangeExceeded ,

Deps , DepsMut ,

Env ,

MessageInfo ,

OverflowError ,

Reply ,

Response ,

StdError ,

StdResult ,

SubMsg , Uint128 ,

WasmMsg , } ;

pub

fn

send_message_to_contract (deps :

DepsMut , amount :

Uint128)

->

Result < Response ,

ContractError

{ let config =

CONFIG . load (deps . storage) ? ;

// Here we define a message to a minimal contract instance to increase the counter by the specified amount let message =

CosmosMsg :: Wasm (WasmMsg :: Execute

{ contract_addr : config . clone ( ) . minimal_contract_address . into_string ( ) , msg :

to_json_binary (& MinimalContractExecuteMsg :: IncreaseCount

{ amount }) ? , funds :

vec! [ ] ,

// Optionally, you can send funds along with the message. }) ;

let current_counter_value :

CurrentValueResponse

= deps . querier . query_wasm_smart (config . minimal_contract_address , & MinimalContractQueryMsg :: CurrentValue

{ } ,) ? ;

// we create a submessage to catch the successfull response Ok (Response :: new ( ) . add_submessage (// Add counter to submsg payload, so we could parse it in the reply handler. // Note that you can also use reply_on_error, as well as reply_always. SubMsg :: reply_on_success (message ,

INCREASE_COUNT_REPLY_ID) . with_payload (to_json_binary (& current_counter_value . current_value) ?) ,) . add_attribute ("action" ,

"send_message_to_contract")) }

/// ----------------------------- REPLY HANDLER ----------------------------------- pub

fn

reply (deps :

DepsMut , _env :

Env , msg :

Reply)

->

StdResult < Response

{ let config =

CONFIG . load (deps . storage) ? ;

// Handle the response message here if msg . id ==

INCREASE_COUNT_REPLY_ID

{ // parse data field from the minimal contract execution response to get the counter value let previous_counter :

Uint128

=

from_json (& msg . payload) ? ;

// make a query to a minimal contract to get current counter value let current_counter_value_via_query :

CurrentValueResponse

= deps . querier . query_wasm_smart (config . minimal_contract_address , & MinimalContractQueryMsg :: CurrentValue

{ } ,) ? ;

// Check whether the counter value was not actually updated by checking the previous counter // value we sent in SubMsg and current counter value from a query if current_counter_value_via_query . current_value <= previous_counter { return

Err (StdError :: generic_err ("counter from SubMsg does not equal to a counter from query" ,)) ; }

Ok (Response :: new ( ) . add_attribute ("reply" ,

"success") . add_attribute ("new_counter" , current_counter_value_via_query . current_value)) }

else

{ Err (StdError :: generic_err ("unknown reply id")) } } In this example,SubMsg is used to capture and handle responses by setting thereply_on attribute. You can also set it toreply_on_error orreply_always depending on the case. In the reply handler, we process the message response based on the IDINCREASE_COUNT_REPLY_ID :

1. First, we check the reply message ID. If it's notINCREASE_COUNT_REPLY_ID
2. , something went wrong, so we return an
3. error.
4. Next, we decode thepayload
5. field of the message to extract theprevious_counter
6. value, which
7. weset in our contract through the execute message
8. .
9. Then, we perform
10. aCurrentValue
11. query to our Minimal contract
12. to get the current counter value from the contract.
13. Finally, we compare these values to ensure everything worked as expected. Simple, right?

Note You can see the whole contracthere.

Modules

With modules, the interaction interface is slightly different, but semantically the same 3 things need to be covered:

1. Sending messages to modules,
2. Processing responses to those messages,
3. Querying data from modules.

Historical background: Stargate and gRPC Historically, some modules had their own WASM bindings implemented by developers to handle messages and queries from smart contracts. For modules without these bindings, Stargate was used to achieve similar functionality. Stargate primarily relied on gRPC for sending and receiving messages, but its interface was inconsistent, returning protobuf-encoded messages for some queries and JSON-encoded ones for others—causing significant frustration for developers. Now that Stargate is outdated, on Neutron, you only need to work with gRPC and protobuf-encoded messages to interact with core modules.

Sending Messages to Cosmos SDK Modules

In this tutorial, we'll explore how a CosmWasm contract interacts with Cosmos SDK modules using custom messages. We'll walk through sending messages to Cosmos SDK modules, handling responses, and querying data. The provided code will guide us through sending tokens (after converting an amount in USD to NTRN using an oracle), handling the response when the tokens are sent, and querying additional data.

Note You can see the whole contracthere. Thesend_tokens function demonstrates how to send tokens by communicating with two Cosmos SDK modules:

- Oracle Module (to get the NTRN price)
- Bank Module (to send tokens)

use

cosmwasm_std :: { entry_point , from_json , to_json_binary ,

Addr ,

Binary ,

CosmosMsg ,

DecimalRangeExceeded ,

Deps , DepsMut ,

Env ,

MessageInfo ,

OverflowError ,

Reply ,

Response ,

StdError ,

StdResult ,

SubMsg , Uint128 ,

WasmMsg , } ;

use

minimal_contract :: contract :: CurrentValueResponse ; use

minimal_contract :: contract :: ExecuteMsg

as

MinimalContractExecuteMsg ; use

minimal_contract :: contract :: QueryMsg

as

MinimalContractQueryMsg ;

use

neutron_std :: types :: cosmos :: bank :: v1beta1 :: MsgSend ; use

neutron_std :: types :: cosmos :: base :: v1beta1 :: Coin

as

SDKCoin ;

use

neutron_std :: types :: slinky :: { self , oracle } ;

pub

fn

send_tokens (deps :

DepsMut , env :

Env , to_address :

String , usd_amount :

Uint128 ,)

->

Result < Response ,

ContractError

{ // get NTRN price from Slinky let slinky_querier =

oracle :: v1 :: OracleQuerier :: new (& deps . querier) ; let ntrn_price = slinky_querier . get_price (Some (slinky :: types :: v1 :: CurrencyPair

{ base :

"NTRN" . to_string ( ) , quote :

"USD" . to_string ( ) , })) ? ;

if ntrn_price . price . is_none ( )

{ return

Err (ContractError :: Std (StdError :: generic_err ("no price for NTRN/USD pair" ,))) ; }

// normalize the price let normalized_price =

Decimal :: from_atomics (Uint128 :: from_str (& ntrn_price . price . unwrap ( ) . price) ? , ntrn_price . decimals as

u32 ,) ? ;

// convert usd amount to ntrn amount let ntrn_amount =

Decimal :: from_str (& usd_amount . to_string ( )) ? . checked_mul (normalized_price) ? . to_uint_floor ( ) ;

// compose bank send message let msg =

MsgSend

{ from_address : env . contract . address . into_string ( ) , to_address , amount :

vec! [SDKCoin

{ denom :

"untrn" . to_string ( ) , amount : ntrn_amount . to_string ( ) , }] , } ;

let sub_msg =

SubMsg :: reply_on_success (Into :: < CosmosMsg

:: into (msg) ,

BANK_SEND_REPLY_ID) ;

// ReplyOn::Success will capture the response only if send message succeeded.

Ok (Response :: new ( ) . add_submessage (sub_msg) . add_attribute ("action" ,

"send_tokens")) } Step 1 : Querying the Oracle Module for the Price of NTRN

Before sending tokens, we need to convert the USD amount into NTRN. To achieve this, the contract queries the Oracle Module for the current NTRN/USD price:

let slinky_querier =

oracle :: v1 :: OracleQuerier :: new (& deps . querier) ; let ntrn_price = slinky_querier . get_price (Some (slinky :: types :: v1 :: CurrencyPair

{ base :

"NTRN" . to_string ( ) , quote :

"USD" . to_string ( ) , })) ? ; * Oracle Module Interaction: * This code uses theoracle::v1::OracleQuerier * to create a querier and request the price * for theNTRN/USD * currency pair from the Slinky oracle module. It uses the get_price method to fetch the current * price. * Error Handling: * If the oracle doesn't return a price (ntrn_price.price.is_none() *), the contract returns an error, * ensuring that no transaction occurs with invalid pricing data.

Step 2 : Normalizing the Price and Converting USD to NTRN

Once the price is retrieved, we need to normalize the result and convert theUSD amount intoNTRN tokens:

let normalized_price =

Decimal :: from_atomics (Uint128 :: from_str (& ntrn_price . price . unwrap ( ) . price) ? , ntrn_price . decimals as

u32 ,) ? ; // convert usd amount to ntrn amount let ntrn_amount =

Decimal :: from_str (& usd_amount . to_string ( )) ? . checked_div (normalized_price) ? . to_uint_floor ( ) ; * The retrieved price is normalized to convert it into a usable format by accounting for the number of decimal places. * The contract multiplies the USD amount by the normalizedNTRN * price to determine how manyNTRN * tokens to send.

Step 3 : Sending Tokens to a Recipient Using the Bank Module

Once the token amount is calculated, the contract sends theNTRN tokens to the recipient by constructing aMsgSend message. This message is sent to the Bank Module:

let msg =

MsgSend

{ from_address : env . contract . address . into_string ( ) , to_address , amount :

vec! [SDKCoin

{ denom :

"untrn" . to_string ( ) , amount : ntrn_amount . to_string ( ) , }] , } ; * Bank Module Interaction: The contract creates aMsgSend * message, which sends the calculatedNTRN * tokens from the * contract's address (env.contract.address *) to the recipient's address (to_address *). * The amount is specified using the micro-denominationuntrn * ;

Step 4 : Handling Success with SubMsg

To handle the response from the Bank Module, the message is wrapped in a SubMsg:

let sub_msg =

SubMsg :: reply_on_success (Into :: < CosmosMsg

:: into (msg) ,

BANK_SEND_REPLY_ID) ; * TheSubMsg::reply_on_success * function ensures that the contract will capture the response when the token transfer * succeeds. TheBANK_SEND_REPLY_ID * helps identify this message's reply when processing the result later.

The function returns a Response object that contains the submessage and logs the action (send_tokens).

Interacting with the contract

Compile the contract binary

Go to thecalling_modules_and_contracts project directory in theonboarding repository:

cd onboarding/calling_modules_and_contracts Then build the contract binary:

docker run --rm -v " (pwd) " :/code \ --mount type = volume,source = "(basename " (pwd) ") _cache",target = /target \ --mount type = volume,source = registry_cache,target = /usr/local/cargo/registry \ --platform linux/amd64 \ cosmwasm/optimizer:0.16.0 cd

.. You will find the compiled binary incalling_modules_and_contracts/artifacts/calling_modules_and_contracts.wasm .

Upload the contract

Now, you need to upload the contract binary (copy thetxhash value from the last line of the command output!):

neutrond tx wasm store calling_modules_and_contracts/artifacts/calling_modules_and_contracts.wasm \ --node tcp://0.0.0.0:26657 --chain-id ntrntest --gas 3000000

\ --fees 100000untrn --from demowallet1 Next, you need to get thecode_id of the binary that you just uploaded:

neutrond q tx 855C2F0D3E120D986B65EB250BBB3C24ED38F7251E928F03DB96AF8186C00973 --output json \ --node tcp://0.0.0.0:26657 | jq ".events[8]" { "type" :

"store_code" , "attributes" :

[{ "key" :

"code_checksum" , "value" :

"5aca867b2af7295c7ff224dd62605a8f1601ccee73161ed41e4c43034327f021" , "index" :

true } , { "key" :

"code_id" , "value" :

"21" , "index" :

true }] } You can see in the output above that thecode_id of our contract binary is21 . Now that we know it, we can finally instantiate our contract (once again, copy thetxhash value):

neutrond tx wasm instantiate 21

'{"minimal_contract_address": "neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj"}' --label minimal_contract \ --no-admin --node tcp://0.0.0.0:26657 --from demowallet1 --chain-id ntrntest \ --gas 1500000 --fees 4000untrn Lets have a look at the transaction arguments and flags once again:

- 21
- : thecode_id
- that we got after uploading our compiled binary. As we mentioned previously, you can instantiate
- multiple identical contracts from onecode_id
- !
- '{"minimal_contract_address": "neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj"}'
- : that's
- ourInstantiateMsg
- that we defined in our contract. If we provided a JSON that
- could not be parsed intoInstantiateMsg
- , we would receive an error.

Now, we need to query the transaction details to get the address of the instantiated contract:

neutrond q tx B45E9D20A5744A81C2C3B0E75D0E740E56E0E0FD20206DDFC77F6FCFE11333B8 --output json --node tcp://0.0.0.0:26657 | jq ".events[8]" { "type" :

"instantiate" , "attributes" :

[{ "key" :

"_contract_address" , "value" :

"neutron1jarq7kgdyd7dcfu2ezeqvg4w4hqdt3m5lv364d8mztnp9pzmwwwqjw7fvg" , "index" :

true } , { "key" :

"code_id" , "value" :

"20" , "index" :

true }] } Congratulations! The contract is now instantiated, and is ready to process our messages.

Contract addresses The address of your contract might be different from what you see in this tutorial. Make sure that you are replacing the addresses from the commands below with the address ofyour contract!

Interact with the contract

Now that the contract is instantiated and we know its address, so we can interact with it.

As you remember, the purpose of our contract is to do two things:

- call the Minimal Contract to increase it's counter;
- send some amount of NTRN tokens to some address.

Let's see how to do each thing:

Increase counter in the Minimal Contract

At first, let's see the counter of the Minimal Contract:

neutrond q wasm contract-state smart neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj \ '{"current_value": {}}' --output json --node tcp://0.0.0.0:26657 { "data" : { "current_value" : "43" } } Let's now increase the value by1 by sending anIncreaseCount message toour contract :

neutrond tx wasm execute neutron1jarq7kgdyd7dcfu2ezeqvg4w4hqdt3m5lv364d8mztnp9pzmwwwqjw7fvg \ '{"increase_count": {"amount": "1"}}' --node tcp://0.0.0.0:26657 --from demowallet1 \ --chain-id ntrntest --gas 1500000 --fees

4000untrn * neutron1jarq7kgdyd7dcfu2ezeqvg4w4hqdt3m5lv364d8mztnp9pzmwwwqjw7fvg * : the address of our instantiated contract. * '{"increase_count": {"amount": "1"}}' * : the JSON representation of theExecuteMsg::IncreaseCount * message that we * defined in our contract.

If we query the Minimal contract once again, we'll see that the current value was increased by 1:

neutrond q wasm contract-state smart neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj \ '{"current_value": {}}' --output json --node tcp://0.0.0.0:26657 { "data" : { "current_value" : "44" } } So our contract did his job successfully!

Send NTRNs to some address

First of all, let's top up our contract with some NTRNs, since it must have something on its address to be able to send tokens to other addresses:

neutrond tx bank send (neutrond keys show demowallet1 -a) neutron1jarq7kgdyd7dcfu2ezeqvg4w4hqdt3m5lv364d8mztnp9pzmwwwqjw7fvg 100000000untrn --node tcp://0.0.0.0:26657 --from demowallet1 \ --chain-id ntrntest --gas 1500000 --fees 4000untrn The purpose of oursend_tokens handler, is to accept a number of tokens in USD (usd_amount), query a price of NTRN token in USD via Slinky, calculated how many NTRNs the contract must send according to the price to some address (to_address).

Let's see NTRN price in USD via Slinky:

curl -X 'GET'

\ 'http://0.0.0.0:1317/slinky/oracle/v1/get_price?currency_pair.Base=NTRN&currency_pair.Quote=USD'

\ -H 'accept: application/json'

| jq

{ "price" :

{ "price" :

"40545945" , "block_timestamp" :

"2024-10-16T13:51:50.926109199Z" , "block_height" :

"15723526" } , "nonce" :

"3411738" , "decimals" :

"8" , "id" :

"43" } Note: you can check a full Slinky's documentatiorhere The response has a lot of fields, but we interested only in a couple of them:

- decimals
-
    - decimals represents the number of decimals that the quote-price is represented in. It is used to scale
- the.price.price
- to its proper value.
- price.price
-
    - represents the quote-price for a token. In our case (taking thedecimals
- field into account), NTRN
- equals to40545945 / 10^8 = ~0.4 USD
- .

That means, if we call our contract's handler withusd_amount = 10 , it'll send 25 NTRNs, cause10 USD * ~0.4 = ~25 NTRN .

Let's check it out!

Let's try to send it to our Minimal Contract instance:

neutrond tx wasm execute neutron1jarq7kgdyd7dcfu2ezeqvg4w4hqdt3m5lv364d8mztnp9pzmwwwqjw7fvg '{"send_ntrn": {"to_address":"neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj", "usd_amount": "1000000"}}' --node tcp://0.0.0.0:26657 --from demowallet1 \ --chain-id ntrntest --gas 1500000 --fees 4000untrn And now let's check balance of the Minimal contract:

neutrond q bank balances neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj --node

tcp://0.0.0.0:26657

balances: - amount: "2500000" denom: untrn pagination: next_key: null total: "0" This is exactly what we expectedPrevious Part 1: Minimal Application Next Part 3: Building a simple Web Application