Part 1: Minimal Application

Overview

Everyone knows how to write smart contracts for Ethereum. Some people even know how to write smart contracts for Solana. But the knowledge of how to write smart contracts for Cosmos SDK chains is sacred, shared by the few, and hard to obtain.

This CosmWasm onboarding tutorial has one goal: to show the reader that building applications with CosmWasm is not scary.

InPart 1 of this tutorial you will learn how to:

1. Create a simple, yet functional smart contract using CosmWasm.
2. Deploy your web3 application locally and on the Neutron testnet.

What are Neutron smart contracts?

We assume you might not knowanything about Neutron, so let's start with a basic overview. If you're already familiar with Cosmos chains, you can skip this section. We've tucked the details into collapsible sections to make it less scary.

Important information: modules and messages Neutron is aCosmos SDK chain, which means that it's a collection ofmodules (bank ,dex , etc.) running on top of theCometBFT consensus. Each module defines a set ofmessages that it can process. A singletransaction can include multiple messages to more than one module.

Tointeract with Neutron , you need tosend messages to Neutron modules. For example, if you want to send someuntrn to your friend, you will need to execute a transaction containing anMsgSend message to thebank module. This message might look like this:

{ "@type" :

"/cosmos.bank.v1beta1.MsgSend" , "from_address" :

"neutron1cvsh2c2vasktkh7krt2w2dhyt0njs0adh5ewqv" , "to_address" :

"neutron1vqwe5hda0sjn0tyhd2w2trk7hktksav2c2hsvc" , "amount" :

[{ "denom" :

"untrn" , "amount" :

"42" }] } As a user, usually you don't have to deal with raw messages yourself — it's done for you either by the CLI or by a UI. You'll examples of using the CLI later on in this document. Important information: smart contracts Smart contracts are enabled on Neutron by a module calledwasmd . As any other module,wasmd defines a set of messages that it can process, e.g.:MsgStoreCode (used to upload compiled contract binaries),MsgExecuteContract (used to execute existing contracts), etc.

For now, the main thing that you need to know about thewasmd messages is thatMsgExecuteContract includes anembedded message for a smart contract :

{ "@type" :

"/cosmwasm.wasm.v1.MsgExecuteContract" , "sender" :

"neutron1cvsh2c2vasktkh7krt2w2dhyt0njs0adh5ewqv" , "contract" :

"neutron1a5xz4zm0gkpcf92ddm7fw8pghg2mf4wm6cyu6cgcruq35upf7auslnnfye" , "msg" :

{ "increase_count" :

{ "amount" :

"42" } } , "funds" :

[ ] } That's becausesmart contract developers can define the messages that the contract is able to process.

In the snippet above, the message under the"msg" key is a message to a smart contract identified by the"contract" address. If you include the message above in a transaction, the following sequence of events will happen:

1. Neutron will identify that the incoming message needs to be sent to thewasmd

2. module.
3. Thewasmd
4. module will look up the contract binary by its address and load it.
5. Thewasmd
6. module will take message under the"msg""
7. key and will pass it to theexecute()
8. entrypoint of
9. the contract.
10. The contract will try to parse the incoming data into the"increase_count"
11. message that is defined on the contract

12. level, and will execute the handler associated with it. Important information: contract lifecycle, contract entry points The creation of a contract involvesthree steps :

13. First you need tocompile

14. the contract binary (more on that in theHow to upload a contract and interact with
15. it?
16. section).
17. Then you need toupload
18. the contract binary to the chain by sending anMsgStoreCode
19. to thewasmd
20. module,
21. which makes Neutron save the binary under a uniquecode_id
22. .
23. Lastly, you need toinstantiate
24. a contract from thiscode_id
25. by sending anMsgInstantiateContract
26. to
27. thewasmd
28. module, which will pass the instantiation message to the contract'sinstantiate()
29. entrypoint and create
30. an actual contract address that you can interact with.

After a contract was instantiated, you can start to send messages to it using theMsgExecuteContract of thewasmd module. Multiple contracts can be created from the samecode_id without the need to re-upload the binary, and each instance with have a unique address.

There are3 main contract entry points that you need to know about that are used by thewasmd module to interact with a contract:instantiate() ,execute() andquery() . We are going to implement all of them in this part of our tutorial.

What does a Neutron smart contract look like?

Areally minimal smart contract would be about 10 lines long, and would be useless for our purposes. Our smart contract is going to be a contract thatactually does something:

1. Keeps aUint128
2. value in the storage.
3. Allows anyone to increase this value by some amount, if the increase amount is less than100
4. .
5. Allows anyone to query the current value from the storage.

Fun, right? Right.Check out the full source code of this contract onGitHub . Take a look if you enjoy diving into raw source code – we've includedlots of comments. But don't worry, we'll walk you through every part of it, step by step, below.

After having a look at the source code, you might have some questions right away, and we will try to address them immediately:

1. "How is this minimal? It's 160 lines of code!"
2. Yes, but 50% of them are comments!
3. "Wait, is this Rust?"
4. Yes, it's in Rust. Rust is scary, but writing CosmWasm smart contracts is probably the
5. easiest
6. thing you can do with Rust, because everything is single-threaded.

The Choice With these initial questions out of the way,you now have a choice :

1. You can proceed directly to theHow to deploy and execute?
2. section at the end of this document to learn how to
3. use CLI to run a Neutron localnet, deploy our example contract and interact with it,

4. Or you can read theHow do I write a smart contract for Neutron?
5. section first if you want to understand the
6. inner workings of our example contract.

How do I write a smart contract for Neutron?

Storage: how to I store data?

TL;DR Usecw_storage_plus::Item andcw_storage_plus::Map types to store standard and custom types.

See it in context!link Storing data is essential, that's why we start with it. Almost any useful contract manages some storage. In CosmWasm, in order to have storage, you need to initialise it using a type from thecw_storage_plus package. In our case, it'sItem :

use

cw_storage_plus :: Item ;

pub

const

COUNTER :

Item < Uint128

=

Item :: new ("counter") ; This will make CosmWasm allocate some space in the persistent storage for a singleUint128 value under the"counter" key. As a smart contract developer, you don't care about this key at all, but you must make sure that each storage item has a unique key.

Saving single integers is kind of lame, but don't worry:you can save almost anything to storage . For example,you can save vectors :Item> .

Oryou can save the types that you created yourself (just make sure you added the fancy derive macro to the type declaration):

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

pub

struct

Config

{ pub important_parameter :

String , }

pub

const

CONFIG :

Item < Config

=

Item :: new ("config") ; For maps,cw_storage_plus has a specialMap type, in which you can also map pretty much anything to anything, if it serialises properly:

pub

const

EXAMPLE_MAP :

Map < Uint128 ,

Uint128

=

Map :: new ("example-map") ; More storage types: cw_storage_plus has even more storage types, some of which allow you to track the height at which a certain value was saved to your storage item. A note on project layout: usually all storage items and storage types are defined in a separate file (src/state.rs), alongside thesrc/contract.rs file. Here we define everything in one place for the sake of simplicity.

Instantiation: how do I initialise a contract?

TL;DR Implement theinstantiate() entrypoint and theInstantiateMsg message to have custom instantiation logic for your contract.

See it in context!link

InstantiateMsg

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

pub

struct

InstantiateMsg

{ initial_value :

Uint128 , } In the snippet above, 2 things happen: the definition ofInstantiateMsg , and the implementation of theinstantiate() entry point.

InstantiateMsg can carry any information we might find useful while populating our new contract. In our case, we decided to useInstantiateMsg to set the initial value of theCOUNTER storage item that was initialised in the previous section.

Note: You need to add the#[derive(Serialize, <...> JsonSchema)] derive macro to the definitions of your custom types (so that they can be properly serialised by Rust). Note: A common practice in the CosmWasm world is to define aConfig type, create a storage item for it and then to set the initial values forConfig parameters in theInstantiateMsg . A note on project layout: usually all messages are defined in a separate file (src/state.rs), alongside thesrc/contract.rs file. Here we define everything in one place for the sake of simplicity.

instantiate()

entrypoint

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

instantiate (deps :

DepsMut , env :

Env , info :

MessageInfo , msg :

InstantiateMsg ,)

->

Result < Response < NeutronMsg

,

ContractError

{ COUNTER . save (deps . storage ,

& msg . initial_value) ? ;

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

"instantiate") . add_attribute ("initial_value" , msg . initial_value) . add_attribute ("contract_address" , env . contract . address) . add_attribute ("sender" , info . sender . to_string ( ))) } Theinstantiate() entry point expects the following arguments:

- deps
- : most importantly, gives you access to thestorage
- and thequerier
- (we'll discuss queries later),
- env
- : keeps information about the execution environment, e.g., the address of the current contract,
- info
- : keeps information about the message that is currently executed, e.g., the address of the message sender,
- msg
- : theInstantiateMsg
- that we just defined.

Note: deps ,env ,info andmsg values are provided automatically by thewasmd module when executing a contract's entrypoint. Most entry points expect a very similar set of arguments, with slight variations. Ourinstantiate() implementation sets the value of theCOUNTER storage item toInstantiateMsg.initial_value . This is our first time saving something to storage, which is quite exciting!

Note: Reading and writing to storage consumes gas, which costs money. Finally, in ourinstantiate() implementation, we addattributes to the successfulResponse . This helps with debugging (we'll cover this in the last section of Part 1). Adding attributes to your response acts like a form of logging. Alternatively, we could just returnOk(Response::new()) , and that would work perfectly fine.

Important information: errors and errors handling The return type of our entrypoint isResult< ContractError> . In simple terms, this means that it can either return a validResponse or aContractError .

If this was really aminimal example, we would not define our own error type, and would simply returnErr(StdError::generic_err("error message")) in case of an error. However, in most cases you want to define contract-specific errors, which we will do in the next section.

You might not have noticed it, but in our implementation of theinstantiate() entrypoint we do some minimal error handling!

Storage operations can potentially fail (although they usually succeed). In our implementation, if the.save() call fails for whatever reason, we immediately propagate the error by putting the? operator at the end of the.save() call. It's also possible to handle errors manually using Rust'smatch operator (seeRust documentation).

Processing messages: how do I make a contract do something?

TL;DR Define all possible messages that your contract needs to process, implement handlers for each of those messages, and match the messages to handlers in theexecute() entrypoint.

See it in context:link

ExecuteMsg

: defining the set of possible actions

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

[serde(rename_all =

"snake_case")] pub

enum

ExecuteMsg

{ IncreaseCount

{ amount :

Uint128

} , } Note:

[serde(rename_all = "snake_case")]

is used to achieve "standard" JSON representation of messages. E.g., the JSON representation of theIncreaseCount message will look like this:{"increase_count": {"amount": "42"}} When you start developing a contract, the first thing that you need to figure out is what actions to you want the contract to perform.

In the snippet above, we define theExecuteMsg enum that has a single variant: theIncreaseCount message, which aUint128 amount field. This means that our contract is going to be able to process only one type of message.

Let's say that upon receiving this message, the contract must increase theCOUNTER storage item value by theamount specified in the message, if theamount is less than 100. Matching this particular logic to theIncreaseCount message type is done in theexecute() entrypoint.

A note on project layout: usually all messages are defined in a separate file (src/msgs.rs), alongside thesrc/contract.rs file. Here we define everything in one place for the sake of simplicity.

execute()

entrypoint: defining handlers for messages

pub

const

MAX_INCREASE_AMOUNT :

Uint128

=

Uint128 :: new (100u128) ;

[derive(Error, Debug, PartialEq)]

pub

enum

ContractError

{ /// Keep access to the StdError, just in case.

[error(transparent)]

Std (

[from]

StdError) ,

/// We will return this error if the user tries to increment the counter by /// more than 100. For no particular reason.

[error(

"Can not increment by more than 100 (got {amount})")] InvalidIncreaseAmount

{ amount :

Uint128

} , }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

execute (deps :

DepsMut , _env :

Env ,

// We don't use Env in our implementation, hence the underscore info :

MessageInfo , msg :

ExecuteMsg ,)

->

Result < Response < NeutronMsg

,

ContractError

{ match msg { ExecuteMsg :: IncreaseCount

{ amount }

=>

execute_increase_amount (deps , info , amount) , } }

pub

fn

execute_increase_amount (deps :

DepsMut , info :

MessageInfo , amount :

Uint128 ,)

->

Result < Response < NeutronMsg

,

ContractError

{ // Return the InvalidIncreaseAmount error if the user tries to increase // by more than 100. We save this value in a well-named constant // MAX_INCREASE_AMOUNT because we are nice people. if amount . gt (& MAX_INCREASE_AMOUNT)

{ return

Err (ContractError :: InvalidIncreaseAmount

{ amount }) ; }

// We need to increase the counter. Step 1: load the current value. // This operation consumes gas! let

mut counter =

COUNTER . load (deps . storage) ? ;

// Step 2: add the user value to the value loaded from the storage. counter += amount ;

// Step 3: save the increased amount to the storage. COUNTER . save (deps . storage ,

& counter) ? ;

Ok (Response :: default ( ) . add_attribute ("action" ,

"execute_add") . add_attribute ("amount" , amount . to_string ( )) . add_attribute ("sender" , info . sender)) } In most contracts, theexecute() entry point doesn't contain specific logic itself; it simply delegates tasks to the appropriate handlers based on the message type. That's exactly what we did in the snippet above: we implemented theexecute_increase_amount() handler and directed our contract to use it when anIncreaseCount message is received.

Note: If the incoming message cannot be parsed into any known message type, an error will be returned. Here are a few important points:

1. We set the maximum allowed increase amount using theMAX_INCREASE_AMOUNT
2. constant, following best practices.
3. We defined theContractError
4. type to return custom errors. TheInvalidIncreaseAmount
5. variant includes
6. theamount
7. parameter to make the error message more informative.
8. If the user tries to increase the counter by more thanMAX_INCREASE_AMOUNT
9. , we return an error. (First time in this
10. tutorial, right?Exciting!
11.)
12. If the user input is valid, we load the current counter value from storage, increase it by the specified amount, and
13. save the new value back to storage.
14. Similar to theinstantiate()
15. implementation, we add some attributes to the response to facilitate easier debugging.

Querying data: how do I... query data?

TL;DR Define all possible query messages that your contract needs to process, implement handlers for each of those messages, and match the messages to handlers in thequery() entrypoint.

See it in contextlink Querying the raw storage data from a smart contract istechnically possible, but is not very convenient. If you want your contract to provide information about it state, you need to (guess what?) implement theQueryMsg and thequery() entrypoint. The process is very similar to what we did with theexecute() entrypoint, the only difference being that in thequery() entrypoint you can not modify the state, and that you don't have theMessageInfo data in it.

[cw_serde]

[derive(QueryResponses)]

pub

enum

QueryMsg

{ /// A query message to get the current value of COUNTER. The #[returns(Uint128)] /// derive marco here is required to generate proper JSON schemas for out smart /// contract.

[returns(Uint128)]

CurrentValue

{ } , }

/// We could simply read the Uint128 value from storage and return it as is, /// but in general it's better to provide a custom response types for your /// queries.

[cw_serde]

pub

struct

CurrentValueResponse

{ pub current_value :

Uint128 , }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps , _env :

Env , msg :

QueryMsg)

->

StdResult < Binary

{ // Similar to execute(), we try to parse msg into any of the known variants of QueryMsg. match msg { QueryMsg :: CurrentValue

{ }

=>

query_current_value (deps) , } }

pub

fn

query_current_value (deps :

Deps)

->

StdResult < Binary

{ let current_value =

& COUNTER . load (deps . storage) ? ; // to_json_binary is a handy helper function from cosmwasm_std that allows you // to convert any properly defined Rust type to StdResult. to_json_binary (& CurrentValueResponse

{ current_value : current_value . clone ( ) , }) } A few things to note here:

- Creating custom response types for your queries is not necessary, but is a good practice, even if you only return a
- single number.
- If our contract was a bit more complicated, we could, of course, create a query message with some query parameters,
- and process the query taking the query parameters into account.
- You are not limited to only reading your own storage values in thequery()
- entrypoint; you can also query other
- contracts, and even modules, if necessary. We'll teach you how to do it inPart 3
- of this tutorial.

A note on project layout: usually all custom response types are defined in a separate file (src/query.rs), alongside thesrc/contract.rs file. Here we define everything in one place for the sake of simplicity.

How to deploy and execute?

Prepare the environment

First of all, you need to install Rusthttps://doc.rust-lang.org/cargo/getting-started/installation.html . After installing Rust, you need to add a wasm target for cargo:

rustup target add wasm32-unknown-unknown Next, install Dockerhttps://docs.docker.com/engine/install/ . You need Docker to build reproducible wasm binaries and to run the localnet.

Create a directory for this tutorial, e.g.:

mkdir neutron-cosmwasm-tutorial-part-1 && cd neutron-cosmwasm-tutorial-part-1 Next, you need to get aneutrond binary. If you don't have Go installed, install Go 1.21 (https://go.dev/dl/).

Clone the Neutron repository and install theneutrond binary:

git clone https://github.com/neutron-org/neutron &&

cd neutron &&

make

install

&&

cd

.. This will put theneutrond binary to yourGOPATH/bin . If you don't haveGOPATH/bin exported yet, export it now to be able to execute the binary conveniently from anywhere on your machine:

export

PATH

PATH : GOPATH /bin Finally, clone the onboarding tutorial repo:

git clone git@github.com:neutron-org/onboarding.git cd onboarding

Run the localnet

First, install thecosmopark tool that allows you to run custom Neutron localnet setups (inputy when prompted to install the package):

npx @neutron-org/cosmopark --help Next, build the images that are required to run a localnet of Neutron consisting of Neutron itself, Gaia, and IBC relayer and an interchain queries relayer. This might be a bit of overkill for our current purposes, but it's good to know how to run a real world setup:

./dockerfiles/build-all.sh Then, start the local network:

npx @neutron-org/cosmopark@latest start localnet_config.json Starting { "level" :30, "time" :1719406723924, "pid" :85823, "hostname" : "Andreis-MBP" , "chain" : "neutron" , "msg" : "Starting ics chain neutron" } < .. .

Done { "level" :20, "time" :1729261511883, "pid" :52118, "hostname" : "Andreis-MBP" , "context" : "main" , "msg" : "cosmopark started" } Note: if you want to shut down the localnet, runnpx @neutron-org/cosmopark@latest stop localnet_config.json . Now you have several containers running on your local machine. One of them ismain-neutron_ics-1 , which is the container running Neutron.

In order to execute any messages, you need to import a key using one of the localnet mnemonics:

neutrond keys add demowallet1 --recover

Enter your bip39 mnemonic

kiwi valid tiger wish shop time exile client metal view spatial ahead

- address: neutron13nfu3ct5xkr0vlswgk3gl9zazp7zan88edz67j name: demowallet1 pubkey: '{"@type":"/cosmos.crypto.secp256k1.PubKey","key":"AqVGLu0hIlfruwPYSddOyDmiy7a2kZ0mJ3Qan8vwzXak"}' type: local Note: you can find more demo mnemonics in thelocalnet_config.json file. Important note: you can also import

your account from a Ledger by runningneutrond keys add my_ledger_account --ledger . Your private key won't be transferred, obviously; to use an account created this way to execute transactions you will always need to specify the--ledger flag in addition to the flags normally provided to a command. To make sure that everything works well, query the balance of your account:

neutrond q bank balances neutron13nfu3ct5xkr0vlswgk3gl9zazp7zan88edz67j --node tcp://0.0.0.0:26657 balances: - amount: "999975000" denom: untrn pagination: next_key: null total: "0"

Compile the contract binary

Go to theminimal_contract project directory in theonboarding repository:

cd onboarding/minimal_contract 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 inminimal_contract/artifacts/minimal_contract.wasm .

Upload the contract

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

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

\ --fees 10000untrn --from demowallet1 Let's see what the command arguments stand for here, becauseit's very important for your general understanding of how Neutron works :

- minimal_contract/artifacts/minimal_contract.wasm
- : path to the compiled contract binary.
- --node tcp://0.0.0.0:26657
- : address of the Neutron node's RPC endpoint. This particular port is exposed by
- themain-neutron_ics-1
- container that we started in the previous section; for mainnet RPC providers, please visit
- themainnet chain registry
- ; for
- testnet RPC providers, please visit
- thetestnet chain registry
- .
- --chain-id ntrntest
- : the chain identifier; any running Cosmos chain has an identifier that you need to provide for
- alltx
- commands. We gave this identifier to our localnet in thelocalnet_config.json
- file. You can find the
- mainnetchain_id
- here
- , and the
- testnetchain_id
- here
- .
- --gas 1500000
- : the gas limit for this transaction; if your gas limit is too low, the transaction will fail. If the
- transaction failed, you can query the transaction details (shown below) to see what amount of gas was actually
- consumed by the transaction.
- --fees 4000untrn
- : the amount that you are ready to pay for executing the transaction; if your fee is too low, the
- transaction will fail. If the transaction failed, you can query the transaction details (shown below) to see what fee
- was actually required to execute the transaction.
- --from demowallet1
- : the account that you are using to sign this transaction.

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" :

"20" , "index" :

true }] } You can see in the output above that thecode_id of our contract binary is19 . Now that we know it, we can finally instantiate our contract (once again, copy thetxhash value):

neutrond tx wasm instantiate 20

'{"initial_value": "42"}' --label minimal_contract \ --no-admin --node tcp://0.0.0.0:26657 --from demowallet1 --chain-id ntrntest \ --gas 3000000 --fees 10000untrn Lets have a look at the transaction arguments and flags once again:

- 20
- : thecode_id
- that we got after uploading our compiled binary. As we mentioned previously, you can instantiate
- multiple identical contracts from onecode_id
- !
- '{"initial_value": "42"}'
- : 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.
- --label minimal_contract
- : every CosmWasm contract needs to have a user-defined label.
- --no-admin
- means that we created our contract without an admin address. The admin address canmigrate
- a contract.

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" :

"neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj" , "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, let's first see whether what that current value of the counter is:

neutrond q wasm contract-state smart neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj \ '{"current_value": {}}' --output json --node tcp://0.0.0.0:26657 { "data" : { "current_value" : "42" } } Important note: we have the number as a string here because that's howUint128 is represented (it's a potentially very big number). When usingUint128 value in JSON, you need to provide them as strings. The current value is42 , which means that our instantiate message did its job. Let's now increase the value by1 by sending anIncreaseCount message to it:

neutrond tx wasm execute neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj \ '{"increase_count": {"amount": "1"}}' --node tcp://0.0.0.0:26657 --from demowallet1 \ --chain-id ntrntest --gas 1500000 --fees 4000untrn *

neutron1nyuryl5u5z04dx4zsqgvsuw7fe8gl2f77yufynauuhklnnmnjncqcls0tj * : 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 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" : "43" } }

Conclusion

In this first part of the tutorial, you learned how to create a simple, yet functional smart contract using CosmWasm, and to deploy and interact with it locally. In the second part of the tutorial, you will learn how to interact with Neutron modules and other smart contracts. Previous CosmWasm + ICQNext Part 2: Calling Modules and Contracts