Introduction to CosmWasm

Let's examine what a smart contract is and the way it works under the hood. The following is a minimal contract that stores a counter value which can be incremented and everyone can query this counter value.

The counter contract template can be found inthis GitHub repository .

Contract basics

A smart contract can be considered an instance of a singleton object whose internal state is persisted on the blockchain. Users can trigger state changes by sending the smart contract JSON messages, and users can also query its state by sending a request formatted as a JSON message.

As a smart contract writer, your job is to define 3 functions that compose your smart contract's interface:

1. instantiate()
2. : a constructor which is called during contract instantiation to provide the initial state
3. execute()
4. : gets called when a user wants to invoke a method on the smart contract, this invokes one of the execute messages defined in the contract underExecuteMsg
5. enum which is essentially a list of transactions the contract supports
6. query()
7. : gets called when a user wants to get data out of a smart contract, this invokes one of the query messages defined in the contract underQueryMsg
8. enum which is a list of queries the contract supports

In our sample counter contract, we will implement one instantiate, one query, and two execute methods.

Contract structure

Following is the file structure of the contract source:

├── Cargo.toml ├── LICENSE ├── NOTICE ├── README.md └── src ├── bin │ └── schema.rs ├── contract.rs ├── error.rs ├── helpers.rs ├── lib.rs ├── msg.rs └── state.rs * Cargo.toml * : Has contract's rust dependencies and some configuration parameters. * src/contract.rs * : Contract entry points such asinstantiate() * ,execute() * andquery() * are defined here. * src/error.rs * : The contract's custom error messages are defined here. * src/lib.rs * : This file defines the list of rust source files that are part of the contract, add the file name here when adding a new one to the contract's source. * src/msg.rs * : Defines all the messages allowed to communicate with the contract. * src/state.rs * : Defines what the structure of the contract's storage will be. * src/bin/schema.rs * : Defines how to generate schema (language-agnostic format of contract messages) from the contract.

Contract state

State handles the state of the database where smart contract data is stored and accessed.

The counter contract has the following basic state, a singleton structState containing:

- count
- , a 32-bit integer with whichexecute()
- messages will interact by increasing or resetting it.
- owner
- , the senderaddress
- of theMsgInstantiateContract
- , which will determine if certain execution messages are permitted.

// src/state.rs

use

schemars :: JsonSchema ; use

cw_storage_plus :: Item ; use

serde :: { Deserialize ,

Serialize } ;

pub

const

STATE :

Item < State

=

Item :: new ("state") ;

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

pub

struct

State

{ pub count :

i32 , pub owner :

Addr } Notice how theState struct holds bothcount andowner . In addition, the derive attribute is applied to auto-implement some useful traits:

- Serialize
- : provides serialization
- Deserialize
- : provides deserialization
- Clone
- : makes the struct copyable
- Debug
- : enables the struct to be printed to string
- PartialEq
- : provides equality comparison
- JsonSchema
- : auto-generates a JSON schema

Addr refers to a human-readable Neutron address prefixed with neutron, e.g.neutron1retp3n0xl0sucqmg4qdk64h7kfflc032xg8na0 .

Messages

InstantiateMsg

TheInstantiateMsg is provided to the contract when a user instantiates a contract on the blockchain through aMsgInstantiateContract . This provides the contract with its configuration as well as its initial state.

Neutron utilizesCosmWasm as the smart contract execution layer. In CosmWasm, the uploading of a contract's code and the instantiation of a contract are regarded as separate events, unlike on Ethereum. This is to allow a small set of vetted contract archetypes to exist as multiple instances sharing the same base code, but be configured with different parameters (imagine one canonical ERC20, and multiple tokens that use its code).

// src/msg.rs

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

pub

struct

InstantiateMsg

{ pub count :

i32 , } The contract creator is expected to supply the initial state in a JSON message. We can see in the message definition below that the message holds one parameter count, which represents the initial count.

{ "count" :

100 }

ExecuteMsg

TheExecuteMsg is a JSON message passed to theexecute() function through aMsgExecuteContract . Unlike theInstantiateMsg , theExecuteMsg can exist as several different types of messages to account for the different types of functions that a smart contract can expose to a user. Theexecute() function demultiplexes these different types of messages to its appropriate message handler logic.

We have twoExecuteMsg :

- Increment
- has no input parameter and increases the value of count by 1.
- Reset
- takes a 32-bit integer as a parameter and resets the value ofcount
- to the input parameter.

// src/msg.rs

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

[serde(rename_all =

"snake_case")] pub

enum

ExecuteMsg

{ Increment

{ } , Reset

{ count :

i32

} , } Any user can increment the current count by 1 using the following message:

{ "increment" :

{ } } Only the owner can reset the count to a specific number. See Logic below for the implementation details.

{ "reset" :

{ "count" :

5 } }

QueryMsg

To support data queries in the contract, you'll have to define both aQueryMsg format (which represents requests), as well as provide the structure of the query's output,CountResponse in this case. You must do this becausequery() will send information back to the user through structured JSON, so you must make the shape of your response known.

// src/msg.rs

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

[serde(rename_all =

"snake_case")] pub

enum

QueryMsg

{ // GetCount returns the current count as a json-encoded number GetCount

{ } , }

// We define a custom struct for each query response

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

pub

struct

CountResponse

{ pub count :

i32 , } Fetch count from the contract using the following message:

{ "get_count" :

{ } } Which should return:

{ "count" :

5 }

Stitching it all together in contract.rs

We've now defined the contract storage, messages the contract can receive.

We now need to specify what method will be called for a given message the contract receives. In other words, what action will be taken when a certain message is received by the contract, which can optionally include updating the state of contract.

Entry points

Entry point defines how an external client can interact with the contract deployed on the chain.

The following entry points are commonly seen in contracts:

instantiate()

Incontract.rs , you will define your first entry-point,instantiate() , or where the contract is instantiated and passed itsInstantiateMsg . Extract the count from the message and set up your initial state where:

- count
- is assigned the count from the message
- owner
- is assigned to the sender of theMsgInstantiateContract

// src/contract.rs

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

instantiate (deps :

DepsMut , _env :

Env , info :

MessageInfo , msg :

InstantiateMsg ,)

->

Result < Response ,

ContractError

{ let state =

State

{ count : msg . count , owner : info . sender . clone ( ) , } ; set_contract_version (deps . storage ,

CONTRACT_NAME ,

CONTRACT_VERSION) ? ; STATE . save (deps . storage ,

& state) ? ;

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

"instantiate") . add_attribute ("owner" , info . sender) . add_attribute ("count" , msg . count . to_string ( ))) }

execute()

This is yourexecute() method, which uses Rust's pattern matching to route the received ExecuteMsg to the appropriate handling logic, either dispatching atry_increment() or atry_reset() call depending on the message received.

// src/contract.rs

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

execute (deps :

DepsMut , _env :

Env , info :

MessageInfo , msg :

ExecuteMsg ,)

->

Result < Response ,

ContractError

{ match msg { ExecuteMsg :: Increment

{ }

=>

try_increment (deps) , ExecuteMsg :: Reset

{ count }

=>

try_reset (deps , info , count) , } }

pub

fn

try_increment (deps :

DepsMut)

->

Result < Response ,

ContractError

{ STATE . update (deps . storage ,

| mut state |

->

Result < _ ,

ContractError

{ state . count +=

1 ; Ok (state) }) ? ;

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

"try_increment")) }

pub

fn

try_reset (deps :

DepsMut , info :

MessageInfo , count :

i32)

->

Result < Response ,

ContractError

{ STATE . update (deps . storage ,

| mut state |

->

Result < _ ,

ContractError

{ if info . sender != state . owner { return

Err (ContractError :: Unauthorized

{ }) ; } state . count = count ; Ok (state) }) ? ; Ok (Response :: new ( ) . add_attribute ("method" ,

"reset")) } Intry_increment() , it acquires a mutable reference to the storage to update the item located at the keystate . It then updates the state's count by returning anOk result with the new state. Finally, it terminates the contract's execution with an acknowledgment of success by returning anOk result with theResponse .

The logic for reset is very similar to increment, except this time, it first checks that the message sender is permitted to invoke the reset function (in this case, it must be the contract owner).

query()

The logic forquery() is similar to that ofexecute() ; however, sincequery() is called without the end-user making a transaction, theenv argument is omitted as no information is necessary.

// src/contract.rs

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps , _env :

Env , msg :

QueryMsg)

->

StdResult < Binary

{ match msg { QueryMsg :: GetCount

{ }

=>

to_binary (& query_count (deps) ?) , } }

fn

query_count (deps :

Deps)

->

StdResult < CountResponse

{ let state =

STATE . load (deps . storage) ? ; Ok (CountResponse

{ count : state . count }) } Note that deps is of typeDeps , notDepsMut (Mut stands for mutable) as in theexecute() , which implies that queries are for read-only operations and do not make any changes to contract's storage. Previous Overview Next CosmWasm + Remix IDE