Title	HOW TO WRITE TERRA SMART CONTRACTS: A COMPLETE TUTORIAL
Description	Tutorial
Date	May 30, 2022
Author	Arashtad
Author URI	https://Arashtad.com

This article explains the connections between the python codes we wrote earlier (same for JavaScript) and the pre-built functions inside the src folder created with the cargo generate–git command. Reading this article will give you a deeper insight into what actually happens when we instantiate, execute or query a state of Terra smart contracts in python code and how it communicates with the CosmWasm libraries written in Rust. Being familiar with Rust programming language can help you better understand the steps we are going to take to interact with the CosmWasm smart contracts on the Terra network. We have provided a small Rust tutorial to make you more familiar with Rust scripts at CosmWasm Smart Contracts - learning Rust series.

WHAT ARE TERRA SMART CONTRACTS?

In the blockchain, Terra smart contracts are instances of a singleton object whose internal state is persisted on the blockchain. A singleton is a class that allows only a single instance of itself to be created and gives access to that created instance. Users can query or set the state changes by sending a JSON message. The details of these messages are going to be explained throughout this article. Terra smart contracts are defined by these 3 main functions:

1. instantiate(): a constructor which is used during contract instantiation to set the initial state.
2. execute(): it is used when a user wants to invoke a method on the smart contract.
3. query(): it is used when a user wants to get data out of a smart contract.

Creating the Template to Write Terra Smart Contracts

The 1st step of creating a smart contract is to create the template using the command below:

```
cargo generate --git https://github.com/CosmWasm/cw-template.git --
branch 0.16 -- name my-first-contract cd my-first-contract
```

The above cargo command creates a structure and boilerplate for creating Terra smart contract. A sample smart contract has the following template, we should define a struct called state and in it, we should determine the count and the address of the owner of the contract. The below code can also be found in the src folder, state.rs file.

```
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};
use cosmwasm_std::Addr;
use cw_storage_plus::Item;
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
pub struct State {
 pub count: i32,
 pub owner: Addr,
}
pub const STATE: Item = Item::new("state");
```

SERIALIZATION & DESERIALIZATION

Notice that some data, like owner and count, can only be persisted as bytes, so we need to convert these data from human-readable format to bytes. That is why we use serialization and deserialization. The CosmWasm team has provided the utility crates such as cosmwasm_storage, which automatically provides serialization and deserialization for commonly used types such as Structs and numbers in Rust. Some of the useful traits applied by deriving attributes in #[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)] are as follows:

1. Serialize: for providing serialization
2. Deserialize: for deserialization
3. Clone: makes the struct copyable
4. Debug: enables the struct to be printed to string
5. PartialEq: provides equality comparison
6. JsonSchema: automatically creates a JSON schema

InstantiateMsg

The instantiateMsg is provided for the users when they want to use MsgInstantiate Contract in their script (Python or JavaScript). This provides the initial state and the configuration of the contract. As opposed to Ethereum, in Terra Smart Contracts, the uploading of a smart contract code and the instantiation of it are considered 2 separate events. The reason for this is to allow different contracts to have the same base code by instantiating. The below code can be found in the src folder and inside the file called msg.rs.

```
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
pub struct InstantiateMsg {
 pub count: i32,
}
```

To create Terra smart contracts, you can send a JSON message like this:

```
{
 "count": 100
}
```

Remember in the python scripts (contracts.py) when we sent the instantiateMsg and set the JSON message to {"count": 15}:

```
contract_address = instantiate_contract(code_id,{"count": 15})
```

When the message InstantiateMsg is sent, the below rust script in the src/contract.rs will extract the message and set up the initial state. In this instantiate function, the count is the extracted count from the message and the address is the address of the account that sent the MsgInstantiateContract message.

```
#[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()))
}
```

EXECUTEMSG

The ExecuteMsg in python or Javascript is the message sent to the execute function in contract.rs through MsgExecuteContract (remember when we imported this in our contract.py). Opposite to what we had in InstantiateMsg, in the ExecuteMsg we have different messages for executing different functionalities(logics). The execute function in the contract.rs file demultiplexes these different messages to their related logic. In the msg.rs file, you will see the definition of the 2 messages Increment and Reset in a public enum.

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum ExecuteMsg {
 Increment {},
 Reset { count: i32 },
}
```

And as you can see, in the contract.rs, Increment and Reset are connected to their appropriate functions in the match msg:

```
ExecuteMsg::Increment {} => try_increment(deps)
Increment {} => try_increment(deps),
```

The complete code of execute function goes like this:

```
#[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),
  }
}
```

Now, the messages that we send in a python or Javascript code, goes like this: increment message:

```
{
 "increment": {}
}
```

reset message:

```
{
 "reset": {
 "count": 5
  }
}
```

remember the kind of script we wrote in python to send the increment message.

```
 execute = execute_contract(deployer,contract_address,{"increment":
{}})
```

Now, let's see how try_increment and try_reset functions work in the contract.rs file. In the try_increment function, the mutable state from state.rs is called. Then, it is updated (incremented) to a new number and then the function returns OK with a new state. Finally the execution is finalized as successful by the OK response.

```
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"))
}
```

The try_reset works the same way as try_increment with the difference that it first checks whether the sender of the message is the owner or not.

```
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"))
}
```

HOW TO INTERACT WITH TERRA SMART CONTRACTS

In the continuation of this journey, we are going to see how the QueryMsg function is defined in Rust in addition to how we can interact with it using Python or JavaScript. QueryMsg: For the query messages, there are two functionalities required:

1. a format of receiving defined in the request GetCount {} inside the QueryMsg enum.
2. And a variable pub count: i32 to respond to the query request which is defined inside the CountResponse struct. The scripts below can be found in the src folder inside the file called 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 {},
}
// Define a custom struct for each query response
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
pub struct CountResponse {
 pub count: i32,
}
```

The following script can be found inside the contract.rs file in the src folder. This script contains 2 functions, one for receiving the query request and two for returning the responding of the state of the contract from the state.rs file. The query function receives the query request and converts the message to binary and passes it to the query_count function which loads the state of the contract from the state.rs and returns the result with the OK response and the message in the { count: state.count } format.

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult {
 match msg {
 QueryMsg::GetCount {} => to_binary(&query_count(deps)?),
  }
}
fn query_count(deps: Deps) -> StdResult {
 let state = STATE.load(deps.storage)?;
 Ok(CountResponse { count: state.count })
}
```

The message we send for querying the state of the contract is in this format.

```
{
 "get_count": {}
}
```

And the respond we get is like the following:

```
{
 "count": 5
}
```

BUILDING TERRA SMART CONTRACT

To build the contract, you can use the 2 commands below:

```
cargo test
cargo wasm
```

The next thing we do is optimize our build by running the following command:

```
cargo run-script optimize
```

To be able to run the above command, you need to make sure you have installed docker.
If you are on arm-64 machine, you should run:

```
docker run --rm -v "$(pwd)":/code \ --mounttype=volume,source="$
(basename
"$(pwd)")_cache",target=/code/target \ --mount
type=volume,source=registry_cache,target=/usr/local/cargo/registry \
cosmwasm/rust-optimizer-arm64:0.12.4
```

And if you are on windows with docker daemon running on WSL1 run the following:

```
docker run --rm -v "$(pwd)":/code \ --mount type=volume,source="$
(basename "$(pwd)")_cache",target=/code/target \ --mount
type=volume,source=registry_cache,target=/usr/local/cargo/registry \
cosmwasm/rust-optimizer-arm64:0.12.4
```

Any of the above commands if run properly (according to the use cases mentioned), should create a new folder called artifacts which contains the my-first-contract.wasm file. To make JSON Schema files, for serialization, you need to register each of the data structures by export_schema(&schema_for!(), &out_dir); you can see the registration in the examples folder, inside the file called schema.rs:

Inside the schema folder, you should be able to see all the requested data structures JSON files.

```
count_response.json
execute_msg.json
instantiate_msg.json
query_msg.json
state.json
```

Inside the state.json, you can see different sections of the .json file, the most important of which is the requires containing the “count” and the “owner”.

```
{ "$schema": "http://json-schema.org/draft-07/schema#", "title":
"State", "type": "object", "required": [ "count", "owner" ],
"properties": { "count": { "type": "integer", "format": "int32" },
"owner": { "$ref": "#/definitions/Addr" } }, "definitions": { "Addr":
{ "description": "A human readable address.\n\nIn Cosmos, this is
typically bech32 encoded. But for multi-chain smart contracts no
assumptions should be made other than being UTF-8 encoded and of
reasonable length.\n\nThis type represents a validated address. It can
be created in the following ways 1. Use `Addr::unchecked(input)` 2.
Use `let checked: Addr = deps.api.addr_validate(input)?` 3. Use `let
checked: Addr = deps.api.addr_humanize(canonical_addr)?` 4.
Deserialize from JSON. This must only be done from JSON that was
validated before such as a contract's state. `Addr` must not be used
in messages sent by the user because this would result in unvalidated
instances.\n\nThis type is immutable. If you really need to mutate it
(Re- ally? Are you sure?), create a mutable copy using `let mut
mutable = Addr::to_string()` and operate on that `String` instance.",
"type": "string" } } }
```

Besides, you can see 2 required sections inside the execute_msg.json:

```
1. "increment" 2. "reset" { "$schema": "http://json-schema.org/draft-
07/schema#", "title": "ExecuteMsg", "anyOf": [ { "type": "object",
"required": [ "increment" ], "properties": { "increment": { "type":
"object" } }, "additionalProperties": false }, { "type": "object",
"required": [ "reset" ], "properties": { "reset": { "type": "object",
"required": [ "count" ], "properties": { "count": { "type": "integer",
"format": "int32" } } } }, "additionalProperties": false } ] }
```

SUMMING UP

we dived deep into the details of the local Terra smart contracts written in Rust or in other words the CosmWasm smart contracts. This type of understanding from scratch helps us interact and communicate better with local Terra and understand what actually happens behind the scenes when we send JSON messages with python scripts to instantiate or execute a smart contract or query data from it. Firstly, we showed how to create a template for initiating state. Then, we created a smart contract and learned how to interact with it. Besides, we learned how to use the QueryMsg function and build a Terra smart contract.

FOLLOW US