Starter guide - Millionaire's Problem

This section provides a detailed breakdown of a Secret smart contract known as the Millionaire's problem . Going through this section won't require you to know Rust (though it would help), but instead we'll focus mostly on the logic, and how different components interact with each other.

Contract Components

We'll start by going through the basic components of a contract.

- Instantiate - the logic that is ran once on initialization of the contract. This will usually be the initial conditions and configuration of the contract.
- Execute - this is where thetransactional
- logic resides. These are functions that modify contract data
- Query - these are functions that are ran as read-only and cannot modify contract data
- State - the long-term storage of the contract. By default, data used by contracts will be lost between transactions.
-

Each contract will contain these basic building blocks, with their logic being built around how they work.

Did you know? Executes require a transaction, which costs gas. Queries are free! This is why it is preferable to use queries when an action does not require modifying data.

Directory Structure

Now we can move on to examining our contract, which solves the Millionaire's problem. We'll start by looking at the directory structure:

For the most part, we can expect the files to contain the following data:

- contract.rs - will contain most of the business logic of the app
- msg.rs - will contain the various interfaces and messages that are used to communicate with the contract
- state.rs - will contain logic pertaining to data structures that are saved in the long-term storage (state)
- cargo.toml - will contain metadata and detail the various libraries that the contract is using
-

Did you know? Contracts for Secret Network depend on forked versions of the standard CosmWasm dependencies

High Level Architecture

Before we jump into the code, we will go over the contract flow. We have to accept inputs from two different users, save their inputs, and expose a way to check which input is greater.

We end up with the following state machine:

Millionaire's Problem State Machine This is a fairly naïve approach, but for the purposes of this example it is fairly straightforward. From looking at this state machine, we can expect that the implementation will contain (at minimum) the following methods:

1. Submit Input
2.
    - allow the user to submit input. We'll also need the user to identify himself somehow so we can print out his identity if he is the winner.
3. Query Result
4.
    - ask the contract which input is greater and return the id of the user whose input was greater
5. Reset
6.
    - resets the state and start over
7.

We'll note thatSubmit Input andReset are both actions that require modifying contract data, so they will beexecutes. Similarly,Query Result is a read-only function, so it can be implemented (as the name suggests) as aquery.

Now let's think about what data our contract will need to store. Eachsubmit input method will require a separate transaction, so we need to save both the user data, and something to help keep track of which step we are on. This means that our state will look something like:

1. User data #1 -
2. User data #2 -

3. Step -
4.

Further Reading: For simplicity we do not differentiate between atransaction and amessage. In reality, the Cosmos-SDK defines each discrete interaction with the chain as amessage. While atransaction is a higher-level object that can contain multiplemessages Alright, now we are armed with all the knowledge we need to dive into the code itself!

Entry Points

Entry points, orhandlers are where messages or queries are handled by the contract - this is basically where all data coming into the contract ends up. We can see the different entry points based on the contract components we described.

Looking at our contract , we can see these entry points defined incontract.rs -

```

Copy

# [entry_point]

pubfninstantiate( deps:DepsMut, _env:Env, _info:MessageInfo, _msg:InstantiateMsg, )->StdResult { //... stuff }

# [entry_point]

pubfnexecute( deps:DepsMut, _env:Env, _info:MessageInfo, msg:ExecuteMsg, )->Result { //... stuff }

# [entry_point]

pubfnquery(deps:Deps, _env:Env, msg:QueryMsg)->StdResult { //... stuff }

```

Instantiate

The instantiate step is where we'll want to create a default contract state with our initial starting conditions to prepare the contract for accepting data. We're not depending on any input from the user in this step, so we can ignore all the function inputs.

Further Reading: The instantiate step is usually where contract ownership is defined, if the contract makes use of admin-only features Let's look at this in the code:

```

Copy

# [entry_point]

pubfninstantiate( deps:DepsMut, _env:Env, _info:MessageInfo, _msg:InstantiateMsg, )->StdResult {

letstate=State::default(); config(deps.storage).save(&state)?;

Ok(Response::default()) }

```

Again, we'll ignore all the scary stuff and look only at lines 9 and 10.

On line #9 we're taking this yet unknownState data type and calling itsdefault() function.

On line #10 we're saving thisState data structure to the contract storage. This means it will be available for us to read the next time the contract is called (from the execute or query entry-points). We can also seedeps.storage is being used as a function parameter. This gives us a clue as to what thedeps variable contains - objects that allow the contract to interact with functionality outside the contract code itself. Some examples of this are using long-term storage, calling efficient crypto APIs, or querying the blockchain state.

Did you know? The instantiate function can only be called once per contract when it is initialized Since we want to

understand what this strangeState structure is, we can sort of guess that there might be some useful information in thestate.rs file. And indeed here we find our definitions:

```

Copy

# [derive(Serialize,Deserialize,Clone,Debug,Default)]

pubstructState{ pubstate:ContractState, pubplayer1:Millionaire, pubplayer2:Millionaire }

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

pubenumContractState{ Init, Got1, Done }

# [derive(Serialize,Deserialize,Clone,Debug,Default,Eq)]

pubstructMillionaire{ name:String, worth:u64 }

```

As per usual, let's ignore all that#[derive] stuff, and instead look at our data structures. We can see that ourState data structure contains the exact types that we expected when thinking about thedesign of the contract. We'll also note that we chose to contain the user data in theMillionaire struct.

Do you even Rust? The #[derive] header tells the compiler to provide implementations for basic behaviour. For example, using _Default _ will tell our compiler to generate the::default() function, which will allocate the structure and set all the values to their defaults (0 for a number, "" for a string, etc.)

Execute

Execute Entry Point

Switching back tocontract.rs we'll just dive right into our execute entry-point, since we already know what to expect

```

Copy

# [entry_point]

pubfnexecute( deps:DepsMut, _env:Env, _info:MessageInfo, msg:ExecuteMsg, )->Result{ matchmsg { ExecuteMsg::SubmitNetWorth{ name, worth }=>try_submit_net_worth(deps, name, worth), ExecuteMsg::Reset{ }=>try_reset(deps), } }

```

Without understanding anything else, we can immediately see our 2 actions that wetalked about previously implemented here. We can also figure out thatmsg is some data type that tells us what function we need to call.

Do you even Rust? Thematch syntax is logically similar toswitch-case that you might be familiar with from other languages

Contract Interface

Toggling over tomsg.rs **** we can see whatExecuteMsg is defined as:

```

Copy

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

# [serde(rename_all="snake_case")]
```

pubenumExecuteMsg{ SubmitNetWorth{ name:String, worth:u64}, Reset{}, }

```

Do you even Rust? In Rust Enums can contain complex data types, which makes them especially useful for interface definitions Yep, as we expect,ExecuteMsg is just an enumeration of the different message types (and their parameters) that a user can send to the contract. The way this works in practice is that the user sends his data as a JSON object. This object then gets parsed and matched according to the definition inExecuteMsg . If the object does not match one of the enum types? The transaction will fail and get rejected!

Do you even Rust? Can you intuitively guess what#[derive(Serialize, Deserialize, PartialEq)] are used for? How about# [serde(rename_all = "snake_case")] ?

Main Functions

Okay, time to zoom back in tocontract.rs and take a look at our functions. The first one we will be looking at istry_submit_net_worth:

```

Copy pubfntry_submit_net_worth(deps:DepsMut, name:String, worth:u64)->Result { letmutstate=config(deps.storage).load()?;

matchstate.state { ContractState::Init=>{ state.player1=Millionaire::new(name, worth); state.state=ContractState::Got1; } ContractState::Got1=>{ state.player2=Millionaire::new(name, worth); state.state=ContractState::Done; } ContractState::Done=>{ returnErr(CustomContractError::AlreadyAddedBothMillionaires); } }

config(deps.storage).save(&state)?;

Ok(Response::default()) }

```

This is the main function that handles the logic of our contract. Although you can probably figure this out easily, we'll describe the logic you're seeing:

1. Read the current state of the contract from long-term storage
2. If we don't have any data, save user data as player#1 and set the state asGot1
3. If we already got the data for the first user, save user data as player#2 and set the state asDone
4. Save the new state in storage.
5.

Super simple. Let's head over to thetry_reset function

```

Copy pubfntry_reset(deps:DepsMut,)->Result { letmutstate=config(deps.storage).load()?;

state.state=ContractState::Init; config(deps.storage).save(&state)?;

Ok(Response::new() .add_attribute("action","reset state")) }

```

At this point I don't even have to explain to you what's going on here. We'll just note that theResponse object in this case returns something called anattribute. Anattribute is a key-value pair that gets returned after a successful message that can help summarizing what happened.Attributes can even be indexed, queried and used as event triggers via WebSocket.

Now we have all the pieces in place. All that remains to be done is to look at how the result is queried by the user.

Query

Let's see what the last contract component we haven't looked at - query - has for us:

```

Copy

[entry_point]

pubfnquery(deps:Deps, _env:Env, msg:QueryMsg)->StdResult { matchmsg {

QueryMsg::WhoIsRicher{}=>to_binary(&query_who_is_richer(deps)?), } }

```

At a glance, we see the same basic structure that we've seen in theExecute entry point. We can also guess thatmsg and theQueryMsg data type are functionally similar to theExecuteMsg type we've seen earlier. By now you'll know where this data type is defined, so you can go take a look at the definition and verify this assumption.

Looking a bit deeper, theto_binary method sticks out as odd. We haven't seen this in our executes or the instantiate function. The reason for this is that queries are returned as binary data. Executes and the instantiate function return complex objects that are saved to the blockchain state, whereas queries are simple and only need to return the specific data the user cares about.

The last piece of the puzzle is thequery_who_is_richer function

```

Copy fnquery_who_is_richer(deps:Deps)->StdResult { letstate=config_read(deps.storage).load()?;

ifstate.state!=ContractState::Done{ returnErr(StdError::generic_err("Can't tell who is richer unless we get 2 data points!")); }

ifstate.player1==state.player2 { letresp=RicherResponse{ richer:"It's a tie!".to_string(), };

returnOk(resp); }

letricher=max(state.player1, state.player2);

letresp=RicherResponse{ // we use .clone() here because ... richer:richer.name().clone(), };

Ok(resp) }

```

Logically, this is super simple. Read the state, check which player has the most money, and return the result.

Do you even Rust? The astute reader will remember thatplayer in this context is actually a complex data structure. How is it possible to callmax(player1, player2) or to check ifplayer1 == player2 ? It turns out you can actually implement the logic for equality and ordering yourself for structs. Head over tostate.rs to see an example of that in action The only thing worth noting is that the response is of the typeRicherResponse (we ignore theStdResult wrapper - it is used to handle and include errors in the possible return types).RicherResponse is a custom type that we defined (inmsg.rs ).

Usually, you will want to define a custom return type for each separate query, which makes data easier to process on the user side - we'll remember that while we talk about user queries for simplicity, in reality, the user will most likely be accessing data through some web application which will be handling both querying the contract and processing the response.

In Closing

That's it! An entire Secret Contract from start to end. Thanks for taking the time to go through all of this guide (or even a small portion of it)! You should now have a good understanding of the building blocks of a contract not only on Secret Network, but for all blockchains that support CosmWasm.

- Intro to Secret Contracts
- 
  - a more in-depth Secret Contract guide
- CosmWasm Documentation
- 
  - everything you want to know about CosmWasm
- Secret.JS
- 
  - Building a web UI for a Secret Contract
- 

Last updated3 months ago On this page *Contract Components * Directory Structure * High Level Architecture * Entry Points * Instantiate * Execute * Query * In Closing

Was this helpful? Edit on GitHub Export as PDF
```