env)

- Example ExecutionMsg
- Response

Was this helpful? Edit on GitHub Export as PDF

Execution Message

An explainer of the Execute file inside of the CosmWasm code framework Execution Messages are contract messages that trigger the execution of a smart contract and perform specific operations, such as updating the contract state or transferring tokens. If you're familiar with RPC or AJAX, you can think of execution messages as the code that runs when a remote procedure is called. On Secret Network, execution messages are usually designed to be functions that run as quickly as possible and exit as early as possible when any errors are encountered, as this will help save gas. You can learn more about contract optimizationhere .

The standard practice on Secret Network is to have anenum with all the valid message types and reject all messages that don't follow the usage pattern dictated in that enum; this enum is conventionally calledExecuteMsg and is usually in a file calledmsg.rs

```

Copy

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

# [serde(rename_all="snake_case")]

pubenumExecuteMsg{ Increment{}, Reset{ count:i32}, }

```

As mentioned earlier, the standard way to choose what the contract should execute is to look at whichExecuteMsg is passed to the function. Let's look at an example.

```

Copy

# [entry_point]

pubfnexecute( deps:&DepsMut, env:Env, msg:ExecuteMsg, )->StdResult{ matchmsg { ExecuteMsg::Increment{}=>try_increment(deps, env), ExecuteMsg::Reset{ count }=>try_reset(deps, env, count), } }

```

In this simple example, the code looks at the message passed, if the passage is theIncrement message, it calls the functiontry_increment , otherwise, if the message is theReset message, it calls the functiontry_reset with the count.

Remember, Rust has implicit returns for lines that don't end in a semicolon, so the result of the two functions above is returned as the result of the execution message. You may have noticed that the execution message takes two arguments in addition to the msg:deps andenv . Let's look at those more closely.

env

As the name perhaps implies,env contains all the information about the environment the contract is running in, but what does that mean exactly? On Secret Network the properties available in the Env struct are as follows:

- block: this contains all the information about the current block. This is the block height (height
-), the current time as a unix timestamp (time
-) and the chain id (chain_id
-) such as secret-4 or pulsar-3.
- message: contains information that was sent as part of the payload for the execution. This is the sender, or the wallet address of the person that called the handle function and sent_funds which contains a vector of native funds sent by the caller (SNIP-20s are not included).
- contract: contains a property with the contract's address.
- contract_code_hash: this is a String containing the code hash of the current contract, it's useful when registering with other contracts such as SNIP20s or SNIP721s or when working on a factory contract.

Now that you have an overview on what all those properties are, let's take a look at a simple Execution Message.

Example ExecutionMsg

```

Copy pubfntry_increment(deps:DepsMut, *env:Env)->StdResult { config(deps.storage).update(|mutstate|->Result*<StdError> { state.count+=1; Ok(state) })?;

deps.api.debug("count incremented successfully"); Ok(Response::default()) }

```

The execution message above reads the state from the storage(Deps) and increments the count property by one, then prints out the new count, if successful.

As mentioned previously, developers don’t really like working with raw binary data, so many resort to using more human friendly ways; you can find out more on the page about storage,here . But for the sake of this example, let’s look at what thatconfig function is doing.

The developers of this contract opted for usingstorage singletons . A storage singleton can be thought as a prefixed typed storage solution with simple-to-use methods. This allows the developer to define a structure for the data that needs to be stored and handles encoding end decoding it, so the developer doesn’t have to think about it. This is how it’s implemented in this contract:

```
Copy usecosmwasm_storage::{singleton, singleton_read,ReadonlySingleton,Singleton};

pubstaticCONFIG_KEY:&[u8]=b"config";

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

pubstructState{ pubcount:i32, pubowner:Addr, }

pubfnconfig(storage:&mutdynStorage)->Singleton { singleton(storage, CONFIG_KEY) }
```

Theconfig function returns a singleton over the config key using theState struct as its type, this means that when reading and writing data from the storage, the singleton automatically serializes or deserializes the State struct.

Response

You may have noticed that the return type for a handle message isResponse . Let's take a look at howResponse is defined:

```
Copy pubstructResponse { /// Optional list of messages to pass. These will be executed in order. /// If the ReplyOn variant matches the result (Always, Success on Ok, Error on Err), /// the runtime will invoke this contract's reply entry point /// after execution. Otherwise, they act like "fire and forget". /// Use SubMsg::new to create messages with the older "fire and forget" semantics. pubmessages:Vec>, /// The attributes that will be emitted as part of a "wasm" event. /// /// More info about events (and their attributes) can be found in [Cosmos SDK docs]. /// /// [Cosmos SDK docs]: https://docs.cosmos.network/main/core/events.html pubattributes:Vec, /// Extra, custom events separate from the mainwasm one. These will have /// wasm- prepended to the type. /// /// More info about events can be found in [Cosmos SDK docs]. /// /// [Cosmos SDK docs]: https://docs.cosmos.network/main/core/events.html pubevents:Vec, /// The binary payload to include in the response. pubdata:Option; }
```

In CosmWasm, theOk result and theResponse object are essential parts of handling contract execution and signaling the outcome to the blockchain. Here's a breakdown of how these work:

Ok Result in CosmWasm

In Rust (and by extension, CosmWasm), functions that interact with the blockchain often return aResult type. This is a way to express whether the function was successful or if it encountered an error.

- Result
- : AResult
- can either be:
-
    - Ok(T)
-
    - : Signifying the function executed successfully and returns a value of typeT
-
    - .
-
    - Err(E)
-
    - : Signifying that an error of typeE
-
    - occurred.

In the context of CosmWasm, theOk result generally signifies that a contract's execution completed successfully, returning aResponse object.

For example:

```
Copy Ok(Response::default())
```

This line means that the contract execution was successful, and it returns a defaultResponse . In CosmWasm, we useStdResultfor contract execution functions. Previous Instantiation Message Next Query Message Last updated1 month ago