

# 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 optimization [here](#).

The standard practice on Secret Network is to have an enum 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 called `ExecuteMsg` and is usually in a file called `msg.rs`

...

Copy

```
[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
[serde(rename_all = "snake_case")]
```

```
pub enum ExecuteMsg {
    Increment {},
    Reset { count: i32 },
}
```

...

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

...

Copy

```
[entry_point]
```

```
pub fn execute(
    deps: &DepsMut,
    env: Env,
    msg: ExecuteMsg,
) -> StdResult {
    match msg {
        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 message is the `Increment` message, it calls the function `try_increment`, otherwise, if the message is the `Reset` message, it calls the function `try_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` and `env`. 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 `ExecuteMsg`

...

```
Copy pub fn try_increment(
    deps: &mutExtern,
    _env: Env,
) -> StdResult {
    config(&mutdeps.storage).update(|mutstate| {
        state.count += 1;
        debug_print!("count = {}", state.count);
        Ok(state)
    })?;
```

```
debug_print("count incremented successfully");
Ok(Response::default())
}
```

...

The execution message above reads the state from the storage and increments the count property by one, then prints out the new count, if successful. But wait — where's the storage hiding, what's `config`? What's `update`?

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 that config function is doing; the developers of this contract, opted for using storage 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 and 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};
```

```
pubstatic CONFIG_KEY:&[u8]=b"config";
```

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

```
pubstruct State { pubcount:i32, }
```

```
pubfn config(storage:&mut S)->Singleton { singleton(storage, CONFIG_KEY) }
```

...

The config function returns a singleton over the config key using the State struct as its type, this means that when reading and writing data from the storage, the singleton automatically serializes or deserializes the State struct. The update method is a shorthand that will load the data, perform the specified action, and store the result in the storage, which makes for a very intuitive experience.

### Response

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

...

```
Copy pubstruct Response where T: Clone + fmt::Debug + PartialEq + JsonSchema, { pub messages: Vec<T>, pub log: Vec<LogAttribute>, pub data: Option<Binary>, }
```

...

It contains a vector of CosmosMsg (messages), an optional vector of type LogAttribute (log), and an optional Binary field (data).

LogAttribute is defined as a simple struct with two String fields: key and value . Binary is just a Vec

CosmosMsg is an enum with a transaction that will be executed once the execution message returns a response.

There are three types of messages:

- BankMsg
  - : sends SCRT from one place to another (such as from the contract's wallet to the caller's wallet)
- StakingMsg
  - : includes messages for Delegating, Redelegating, Undelegating, and withdrawing rewards to/from a delegator.
- WasmMsg
  - : includes two messages: Execute
    - to execute another contract, and Instantiate
      - to instantiate a contract.

Last updated 6 months ago On this page \* [env](#) \* [Example ExecutionMsg](#) \* [Response](#)

Was this helpful? [Edit on GitHub](#) [Export as PDF](#)