Query

Querying is the other half of the coin to messages. You can think of queries as a database read, or a way of querying state.

Generally you will find the available query messages inmsg.rs orquery.rs , depending on how the contract author has structured the code.

You can query via an external client (over API or via CLI), or an internal client (within a contract, to another contract). Some of the finer details of how this works can be found in theQuerying Architecture section .

Most queries you use will be custom queries. These access the contract's data store in read-only mode. These queries can look up data and perform additional computation or processing as needed. As a result, a gas limit is enforced on these queries.

Custom queries consist of an entry in theQueryMsg enum, and are handled in the contract'squery function.

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

[serde(rename_all =

"snake_case")] pub

enum

QueryMsg

{ // ResolveAddress returns the current address that the name resolves to ResolveRecord

{ name :

String

} , Config

{ } , } You can find the code for this example in contexthere .

The contract then handles this in thequery function:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps , env :

Env , msg :

QueryMsg)

->

StdResult < Binary

{ match msg { QueryMsg :: ResolveRecord

{ name }

=>

query_resolver (deps , env , name) , QueryMsg :: Config

{ }

=>

to_binary (& config_read (deps . storage) . load ( ) ?) , } } Wherequery_resolver is just another function, andconfig_read is a helper that wraps access to the data store.

The custom queries are exposed viathe query function . Previous Entry points Next Events