Testing

At some point in Smart Contract development, everyone will arrive at tests. Tests are what give you and your protocol sleep at night and ensure changes can be rapidly deployed to the contracts codebase without breaking everything else.

A great set of contracts will have a great set of tests generally divided into two areas of testing. Unit testing and Integration testing.

Unit Testing

See here for aguide on unit testing

Integration Testing withcw-multi-test

The cw-multi-test package offered in the cw-plus repo provides an interesting way to test your smart contracts without going all the way to deploying them on a testnet. Before using multi-test the flow to me was to have some pipeline that would set up your contracts on a given chain (maybe testnet, maybe local) perform some tests, and then if possible destroy/self-destruct the contracts.

All of that can be taken away almost in preference for cw-multi-test-based integration tests which enable you to test the flows and interactions between smart contracts. There is still a place for the flow described above but I have had a better experience writing these integration tests once you figure out the intricacies of multi-test. I hope to clear some of those intricacies up here with some tips, resources, and steps.

cw-multi-test

concepts

There are a few main concepts to understand before you will be able to simulate a blockchain environment in Rust and run tests that involve contract -> contract, and contract -> bank interactions.

In this section we will take a step-by-step look through writing a test with cw-multi-test, explaining some important concepts along the way. To start we need a specimen contract such as thecw-template which is a simple boilerplate contract containing two functions:Increment andReset .

We start as we always start with a new test file with a few imports:

use

cosmwasm_std :: testing :: { mock_env ,

MockApi ,

MockQuerier ,

MockStorage ,

MOCK_CONTRACT_ADDR } ; use

cw_multi_test :: { App ,

BankKeeper ,

Contract ,

ContractWrapper } ; The above imports will give us a wide palette of tools to start crafting a test. The first import to look at here isApp which will become the simulated blockchain environment in which our tests will be executed.

App

The main entry point to the system is calledApp , which represents a blockchain app. It maintains an idea of block height and time, which you can update to simulate multiple blocks. You can useapp.update_block(next_block) to increment timestamp by 5s and height by 1 (simulating a new block) or you can write any other mutator to advance more.

It exposes an entry pointApp.execute that allows us to execute anyCosmosMsg and it wraps it as an atomic transaction. That is, only ifexecute returns success, will the state be committed. It returns the data and a list of Events on successful

execution or anErr(String) on error. There are some helper methods tied to theExecutor trait that create theCosmosMsg for you to provide a less verbose API.instantiate_contract ,execute_contract , andsend_tokens are exposed for your convenience in writing tests. Each executes oneCosmosMsg atomically as if it was submitted by a user. (You can also useexecute_multi if you wish to run multiple messages together that revert all state if any fail).

The other key entry point toApp is theQuerier interface that it implements. In particular, you can useApp.wrap() to get aQuerierWrapper , which provides all kinds of nice APIs to query the blockchain, likeall_balances andquery_wasm_smart . Putting this together, you have oneStorage wrapped into an application, where you can execute contracts and bank, query them easily, and update the currentBlockInfo , in an API that is not very verbose or cumbersome. Under the hood, it will process all messages returned from contracts, move "bank" tokens, and call into other contracts.

You can create an App for use in your test code like:

fn

mock_app ( )

->

App

{ let env =

mock_env ( ) ; let api =

Box :: new (MockApi :: default ( )) ; let bank =

BankKeeper :: new ( ) ;

App :: new (api , env . block , bank ,

Box :: new (MockStorage :: new ( ))) }

Mocking contracts

Mocking your contracts is one of the mantras of multi-test but also one of the main obstacles to getting yourself a working test. First consider that whatever contract you want to test needs to be either mocked or wrapped up.cw-multi-test provides theContractWrapper which allows you to wrap up the logical pieces of your contract (instantiate, executors, queries) and deploy it to a mocked network.

Mocking all your contracts and then testing one can be done in a scripting fashion but for maintainability, I recommend trying to define all your wrapped contracts as functions so you can reuse them:

use

crate :: contract :: { execute , instantiate , query , reply } ;

pub

fn

contract_stablecoin_exchanger ( )

->

Box < dyn

Contract < Empty

{ let contract =

ContractWrapper :: new_with_empty (execute , instantiate , query ,) . with_reply (reply) ; Box :: new (contract) } The above is a more complex example but let's break it down real quick. We import the execute, instantiate, query, and reply functions that are used at runtime by the contract and then make our wrapper from them to be used in the tests.

To reply or not reply

Depending on the makeup of your contract, when you go to create a ContractWrapper you may not needwith_reply if your contract does not implement areply function. After mocking out a contract, two more steps follow which are; storing the code and then setting up a contract from the code object. You will notice this is the exact same process for deploying to a testnet or mainnet chain whereas in unit tests you work with a mocked_env, usingmock_dependencies and passing inmock_info .

Storing and Instantiating a Contract:

Before a contract can be instantiated in acw-multi-test environment, the contract first has to be stored. Once stored the contract can be instantiated using its associatedcode_id

let contract_code_id = router . store_code (contract_stablecoin_exchanger ( )) ; Instantiating from the new code object:

let mocked_contract_addr = router . instantiate_contract (contract_code_id , owner . clone ( ) ,

& msg ,

& [ ] ,

"super-contract" ,

None) . unwrap ( ) ; All the above gives you 1 mocked contract. As you start to test you may see errors like

- No ContractData
- Contract " does not exist

If you get any of these there's a good chance you a missing a mock. When in multi-test land, everything you interact with that can be considered a contract needs to be mocked out. That includes your own simple little utility contract you don't intend to test right now as well as any services your contract interacts with.

Look at your contract and see if you are passing in any dummy contract addresses, that's the most likely cause. If you find any you must; mock it out with the above method; instantiate it with the above method; capture the address and pass that instead of a dummy one. Took me a while to get a complex contract fully mocked out but hopefully, this helps you. Now for the next glaring problem I faced. Mocking other services!!

info An address must be lowercase alphanumeric to be considered valid. For example, "owner" is valid but "OWNER" is not.

Putting it all together

use

cosmwasm_std :: testing :: { mock_env ,

MockApi ,

MockQuerier ,

MockStorage ,

MOCK_CONTRACT_ADDR } ; use

cw_multi_test :: { App ,

BankKeeper ,

Contract ,

ContractWrapper } ; use

crate :: contract :: { execute , instantiate , query , reply } ; use

crate :: msg :: { InstantiateMsg ,

QueryMsg }

fn

mock_app ( )

->

App

{ let env =

mock_env ( ) ; let api =

Box :: new (MockApi :: default ( )) ; let bank =

BankKeeper :: new ( ) ;

App :: new (api , env . block , bank ,

Box :: new (MockStorage :: new ( ))) }

pub

fn

contract_counter ( )

->

Box < dyn

Contract < Empty

{ let contract =

ContractWrapper :: new_with_empty (execute , instantiate , query ,) ; Box :: new (contract) }

pub

fn

counter_instantiate_msg (count :

Uint128)

->

InstantiateMsg

{ InstantiateMsg

{ count : count } }

[test]

fn

counter_contract_multi_test ( )

{ // Create the owner account let owner =

Addr :: unchecked ("owner") ; let

mut router =

mock_app ( ) ;

let counter_contract_code_id = router . store_code (contract_counter ( )) ; // Setup the counter contract with an initial count of zero let init_msg =

InstantiateMsg

{ count :

Uint128 :: zero ( ) } // Instantiate the counter contract using its newly stored code id let mocked_contract_addr = router . instantiate_contract (counter_contract_code_id , owner . clone ( ) ,

& init_msg ,

& [ ] ,

"counter" ,

None) . unwrap ( ) ;

// We can now start executing actions on the contract and querying it as needed let msg =

ExecuteMsg :: Increment

{ } // Increment the counter by executing the prepared msg above on the previously setup contract let _ = router . execute_contract (owner . clone ( ) , mocked_contract_addr . clone ( ) , & msg , & [ ] ,) . unwrap ( ) ; // Query the contract to verify the counter was incremented let config_msg =

QueryMsg :: Count { } ; let count_response :

CountResponse

= router . wrap ( ) . query_wasm_smart (mocked_contract_addr . clone ( ) ,

& config_msg) . unwrap ( ) ; asserteq! (count_response . count ,

1)

// Now let's reset the counter with the other ExecuteMsg let msg =

ExecuteMsg :: Reset

{ } let _ = router . execute_contract (owner . clone ( ) , mocked_contract_addr . clone ( ) , & msg , & [ ] ,) . unwrap ( ) ;

// And again use the available contract query to verify the result // Query the contract to verify the counter was incremented let config_msg =

QueryMsg :: Count { } ; let count_response :

CountResponse

= router . wrap ( ) . query_wasm_smart (mocked_contract_addr . clone ( ) ,

& config_msg) . unwrap ( ) ; asserteq! (count_response . count ,

0) }

Mocking 3rd party contracts

If you read the above section you will have a gist of the amount of setup work you will have to do by mocking out your contracts as your mocking and trying to progress with a test you may get caught up when you realize your contracts interact with Terraswap, Anchor or some other service out in the IBC. No biggie right?

You'll start just trying to mock out one of these services in the exact same way as we did above only to realize, wait, we need access to the code.. the contract code is what we import to getexecute, instantiate, query . But then you notice protocols don't include their contract code in their rust packages! They only include what you need to interact with them i.e msgs and some helpers.

All hope is not lost however you can still progress by trying to make a thin mock of whatever service you interact with. The process of doing so is similar to what you will do with mocking your contracts (described above) except you will need to fill in all the functionality. This is made easier because you can also a smaller ExecuteMsg with only the func you use or a MockQuery handler with only the queries for example. Here is an example of our mock third-party contract:

pub

fn

contract_ping_pong_mock ( )

->

Box < dyn

Contract < Empty

{ let contract =

ContractWrapper :: new (| deps , _ , info , msg :

MockExecuteMsg |

->

StdResult < Response

{ match msg { MockExecuteMsg :: Receive (Cw20ReceiveMsg

{ sender : _ , amount : _ , msg , })

=>

{ let received :

PingMsg

=

from_binary (& msg) ? ; Ok (Response :: new ( ) . add_attribute ("action" ,

"pong") . set_data (to_binary (& received . payload) ?)) } } }) }

| _ , _ , msg :

MockQueryMsg |

->

StdResult < Binary

{ match msg { MockQueryMsg :: Pair

{ }

=>

Ok (to_binary (& mock_pair_info ( )) ?) , You get a lot of flexibility when you are defining your own mocked contract. You can throw away things like deps, env, info with_ if you never use them and return any responses you want for a given execute msg or query. The challenge then becomes how do I mock out all these services? Seecw-terra-test-mocks for some Terra-based mocks.

Platform Specific Variations

Different chains and hubs in the Cosmos ecosystem may have some variations on how migrations are done on their respective networks. Refer to their respective pages and discord for more information. Previous Code Pinning Next Sudo Execution * Unit Testing * Integration Testing withcw-multi-test * cw-multi-test concepts * * App * * Mocking contracts * * Storing and Instantiating a Contract: * * Putting it all together * Platform Specific Variations