# Unit Tests

How to write Unit Tests for Secret Contracts

Writing Secret Contract Unit Tests

To ensure that Secret smart contract code is reliable and free of errors, it's important to test it thoroughly. One effective way to achieve this is throughunit testing.

Unit testing involves breaking down a program into its smallest components or units and testing each one in isolation. By testing each unit separately, developers can pinpoint the root cause of any errors or bugs and fix them quickly.

In Rust, unit testing is supported by theRust testing framework , which provides a set of macros and utilities for writing and running tests. Rust's testing framework is built into the standard library and is designed to:

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you expect.
4.

Secret Contracts utilize the Rust testing frameworkas well as additional cosmwasm-std utilities , such asmock_dependencies andmock_env functions.

We will explore the basics of unit testing for Secret Contracts, including how to write and run tests, organize test code, and use test-driven development (TDD) to ensure code quality. With this foundation, you'll be well on your way to writing robust and reliable Secret contracts.

Testing a Secret Counter Contract

For a practical understanding of writing unit tests for Secret Contracts, navigate to thecontract.rs file of the Secret Counter template . Here you will find 3 unit tests:

- proper_initialization()
- increment()
- reset()
-

Let's examineproper_initialization() to understand how unit tests check for correctness of various execution messages in Secret Smart Contracts.

Deep dive: Understanding the proper-initialization() test

Review line 71 of the contract.rs file, which contains theproper_initialization() test and tests whether or notthe counter contract is properly instantiated . Let's break this function down line-by-line so we have a thorough understanding of each piece of the code.

```

Copy //examine this code block and then read the analysis below

# [cfg(test)]

mod tests { use super::*; use cosmwasm_std::testing::*; use cosmwasm_std::{from_binary, Coin, StdError, Uint128}; }

```

Theuse super::*; line imports all the modules from the parent module ie everything that is imported at the topof thecontract.rs file). This allows the test module to accesscontract.rs 's imports and test its functionality.

Theuse cosmwasm_std::testing::*; line imports all the testing utilities from thecosmwasm_std crate.

The#[cfg(test)] annotation on the tests module tells Rust to compile and run this test code only when you runcargo test , and not when you runcargo build . This saves compile time when you only want to build the library and saves space in the resulting compiled artifact because the tests are not included.

Themod tests { } block defines a new module namedtests . This is a conventional way of organizing test functions in Rust. Tests can be run withcargo test .

```
```

Copy cargotest

```

If you runcargo test , the terminal will return 3 passing tests! Since Rust's testing framework, Cargo, runs tests in parallel by default, this can lead to nondeterministic behavior if the code being tested is not designed to handle concurrent execution safely (for example,keymap withiterator andAppendStore ).The immediate fix for this issue is to enforce the tests to run serially ,thus avoiding the problems caused by concurrent access and modification . This can be achieved by using the following command:

```

Copy cargotest----test-threads=1

```

This command tells Cargo to run the tests with only one thread, effectively serializing test execution and preventing concurrent access to shared resources.

Mock dependenices

```

Copy

# [test]

fn proper_initialization() { let mut deps = mock_dependencies(); let info = mock_info( "creator", &[Coin { denom: "earth".to_string(), amount: Uint128::new(1000), }], );

```

Thelet mut deps = mock_dependencies(); line creates a new set ofmock dependencies , which simulate the necessary dependencies of a smart contract in a testing environment. These dependencies include storage, a message handler, and an API client.

Thelet info = mock_info("creator", &[Coin { denom: "earth".to_string(), amount: Uint128::new(1000), }], ); line creates a new mock transaction context, which simulates the context in which a transaction would be executed on the blockchain. This context includes information about the sender of the transaction (in this case, the creator), as well as any tokens (in this case, a single "earth" token with a balance of 1000).

Taken together, these lines set up a mock environment in which the counter contract can be tested. Theproper_initialization() function can now perform its tests using these mock dependencies and transaction context.

Init_msg

```

Copy let init_msg = InstantiateMsg { count: 17 };

let res = instantiate(deps.as_mut(), mock_env(), info, init_msg).unwrap();

assert_eq!(0, res.messages.len());

let res = query(deps.as_ref(), mock_env(), QueryMsg::GetCount {}).unwrap();

let value: CountResponse = from_binary(&res).unwrap();

assert_eq!(17, value.count); }

```

Thelet init_msg = InstantiateMsg { count: 17 }; line creates a newInstantiateMsg struct with an initial count value of 17. This message is used to initialize the counter contract's state.

Remember,we defined an instantiate message in msg.rs, which is why this struct is required to instantiate the contract. If there was no instantiation struct with a starting count, the test would fail. Thelet res = instantiate(deps.as_mut(), mock_env(), info, init_msg).unwrap(); line instantiates the smart contract with the given dependencies that we defined above, as well as the transaction context, and instantiation message. This initializes the smart contract's state and returns aResponse struct that contains any messages that the contract emits during initialization.
```

Theassert_eq!(0, res.messages.len()); line checks that no messages were emitted during smart contract instantiation. If any messages were emitted, this assertion would fail and the test would panic.

Thelet res = query(deps.as_ref(), mock_env(), QueryMsg::GetCount {}).unwrap(); line queries the smart contract's state using theQueryMsg::GetCount message. This retrieves the current count value stored in the smart contract's state.

Thelet value: CountResponse = from_binary(&res).unwrap(); line deserializes the query response data (which is in binary format) into aCountResponse struct. This struct represents the result of the query and contains the current count value.

Finally, theassert_eq!(17, value.count); line checks that the current count value retrieved from the smart contract's state is equal to the expected value of 17. If the current count value is not equal to 17, this assertion would fail and the test would panic.

## Next Steps

With this information, examine the other testing functions in the counter contract. You should have all of the resources you need to start writing your very own unit tests! Now let's learn about Multitests