Integration Tests

Unit Tests vs. Integration Tests

Unit tests are great for ensuring that functionality works as expected at an insolated, functional-level. This might include checking that functionget_nth_fibonacci(n: u8) works as expected, handles invalid input gracefully, etc. Unit tests in smart contracts might similarly test public functions, but can get unruly if there are several calls between accounts. As mentioned in theunit tests section, there is aVMContext object used by unit tests to mock some aspects of a transaction. One might, for instance, modify the testing context to have thepredecessor_account_id of"bob.near" . The limits of unit tests become obvious with certain interactions, like transferring tokens. Since"bob.near" is simply a string and not an account object, there is no way to write a unit test that confirms that Alice sent Bob 6 NEAR (Ⓝ). Furthermore, there is no way to write a unit test that executes cross-contract calls. Additionally, there is no way of profiling gas usage and the execution of the call (or set of calls) on the blockchain.

Integration tests provide the ability to have end-to-end testing that includes cross-contract calls, proper user accounts, access to state, structured execution outcomes, and more. In NEAR, we can make use of theworkspaces libraries in bothRust andJavaScript for this type of testing on a locally-run blockchain or testnet.

When to Use Integration Tests

You'll probably want to use integration tests when:

- There are cross-contract calls.
- There are multiple users with balance changes.
- You'd like to gather information about gas usage and execution outcomes on-chain.
- You want to assert the use-case execution flow of your smart contract logic works as expected.
- You want to assert given execution patterns do not work (as expected).

Setup

Unlike unit tests (which would often live in thesrc/lib.rs file of the contract), integration tests in Rust are located in a separate directory at the same level as/src , called/tests (read more). Refer to this folder structure below:

├── Cargo.toml ← contains dependencies for contract and dev-dependencies for workspaces-rs tests ├── src │ └── lib.rs ← contract code ├── target └── tests ← integration test directory └── integration-tests.rs ← integration test file info These tests don't have to be placed in their own/tests directory. Instead, you can place them in the/src directory which can be beneficial since then you can use the non-exported types for serialization within the test case. A sample configuration for this project'sCargo.toml is shown below:

[package] name = "fungible-token-wrapper" version = "0.0.2" authors = ["Near Inchello@nearprotocol.com"] edition = "2021"

[dev-dependencies] anyhow = "1.0" near-primitives = "0.5.0" near-sdk = "4.0.0" near-units = "0.2.0" serde_json = "1.0" tokio = { version = "1.14", features = ["full"] } workspaces = "0.4.1"

remember to include a line for each contract

fungible-token = { path = "./ft" } defi = { path = "./test-contract-defi" }

[profile.release] codegen-units = 1

Tell rustc to optimize for small code size.

opt-level = "z" lto = true debug = false panic = "abort" overflow-checks = true

[workspace]

remember to include a member for each contract

members = ["ft", "test-contract-defi",] Theintegration-tests.rs file above will contain the integration tests. These can be run with the following command from the same level as the testCargo.toml file:

cargo test --test integration-tests

Comparing an Example

Unit Test

Let's take a look at a very simple unit test and integration test that accomplish the same thing. Normally you wouldn't duplicate efforts like this (as integration tests are intended to be broader in scope), but it will be informative.

We'll be using snippets from thefungible-token example from thenear-sdk-rs repository to demonstrate simulation tests.

First, note this unit test that tests the functionality of thetest_transfer method:

examples/fungible-token/ft/src/lib.rs loading ... See full example on GitHub The test above sets up the testing context, instantiates the test environment throughget_context() , calls thetest_transfer method, and performs thestorage_deposit() initialization call (to register with the fungible token contract) and theft_transfer() fungible token transfer call.

Let's look at how this might be written with workspaces tests. The snippet below is a bit longer as it demonstrates a couple of things worth noting.

Workspaces Test

examples/fungible-token/tests/workspaces.rs loading ... See full example on GitHub In the test above, the compiled smart contract.wasm file (which we compiled into the/out directory) for the Fungible Token example is dev-deployed (newly created account) to the environment. Theft_contract account is created as a result from the environment which is used to create accounts. This specific file's format has only one test entry point (main), and every test is declared with#[tokio::test] . Tests do not share state between runs.

Notice the layout withintest_total_supply ..call() obtains its required gas from the account performing it. Unlike the unit test, there is no mocking being performed before the call as the context is provided by the environment initialized duringinit() . Every call interacts with this environment to either fetch or change state.

info Pitfall : you must compile your contract before running integration tests. Because workspaces tests use the.wasm files to deploy the contracts to the network. If changes are made to the smart contract code, the smart contract wasm should be rebuilt before running these tests again. note In case you wish to preserve state between runs, you can call multiple tests within one function, passing the worker around from aworkspaces::sandbox() call.

Helpful Snippets

Create an Account

integration-tests/rs/src/tests.rs loading ... See full example on GitHub note You can also create adev_account without having to deploy a contract as follows:

workspaces/tests/create_account.rs loading ... See full example on GitHub

Create Helper Functions

integration-tests/rs/src/helpers.rs loading ... See full example on GitHub

Spooning - Pulling Existing State and Contracts from Mainnet/Testnet

This example showcases spooning state from a testnet contract into our local sandbox environment:

examples/src/spooning.rs loading ... See full example on GitHub For a full example, see theexamples/src/spooning.rs example.

Fast Forwarding - Fast Forward to a Future Block

workspaces testing offers support for forwarding the state of the blockchain to the future. This means contracts which require time sensitive data do not need to sit and wait the same amount of time for blocks on the sandbox to be produced. We can simply just callworker.fast_forward to get us further in time:

examples/src/fast_forward.rs loading ... See full example on GitHub For a full example, take a look atexamples/src/fast_forward.rs .

Handle Errors

integration-tests/rs/src/tests.rs loading ... See full example on GitHub note ReturningErr(msg) is also a viable (and arguably simpler) implementation.

Batch Transactions

Batch Transaction - workspace-rs let res = contract . batch (& worker) . call (Function :: new ("ft_transfer_call") . args_json ((defi_contract . id ( ) , transfer_amount ,

Option :: < String

:: None ,

"10")) ? . gas (300_000_000_000_000

/

2) . deposit (1) ,) . call (Function :: new ("storage_unregister") . args_json ((Some (true) ,)) ? . gas (300_000_000_000_000

/

2) . deposit (1) ,) . transact ( ) . await ? ;

Inspecting Logs

Logs - workspaces-rs assert_eq! (res . logs ( ) [1] , format! ("Closed @{} with {}" , contract . id ( ) , initial_balance .0

- transfer_amount .0)) ; Examining receipt outcomes:

Logs - workspaces-rs let outcome =

& res . receipt_outcomes ( ) [5] ; assert_eq! (outcome . logs [0] ,

"The account of the sender was deleted") ; assert_eq! (outcome . logs [2] ,

format! ("Account @{} burned {}" , contract . id ( ) ,

10)) ;

Profiling Gas

CallExecutionDetails::total_gas_burnt includes all gas burnt by call execution, including by receipts. This is exposed as a surface level API since it is a much more commonly used concept:

Gas (all) - workspaces-rs println! ("Burnt gas (all): {}" , res . total_gas_burnt) ; If you do actually want gas burnt by transaction itself you can do it like this:

Gas (transaction) - workspaces-rs println! ("Burnt gas (transaction): {}" , res . outcome ( ) . gas_burnt) ; If you want to see the gas burnt by each receipt, you can do it like this:

Gas (receipt) - workspaces-rs for receipt in res . receipt_outcomes ( )

{ println! ("Burnt gas (receipt): {}" , receipt . gas_burnt) ; } Edit this page Last updatedonJan 31, 2024 bygagdiez Was this page helpful? Yes No

Previous Reproducible Builds Next Unit Tests