Integration Tests

Integration tests enable to deploy your contract in the NEARtestnet or a localsandbox , and create test-users to interact with it. In this way, you can thoroughly test your contract in a realistic environment.

Moreover, when using the localsandbox you gain complete control of the network:

1. Create testAccounts
2. and manipulate theirState
3. andBalance
4. .
5. Simulate errors on callbacks.
6. Control the time-flow and fast-forward into the future (Rust ready, TS coming soon).

NEAR Workspaces In NEAR, integration tests are implemented using a framework calledWorkspaces . Workspaces comes in two flavors:Rust and Typescript .

All of ourexamples come with integration testing.

Snippet I: Testing Hello NEAR

Lets take a look at the test of ourQuickstart Project Hello NEAR , where we deploy the contract on an account and test it correctly retrieves and sets the greeting.

- JavaScript

contract-ts/sandbox-ts/main.ava.ts loading ... See full example on GitHub

Snippet II: Testing Donations

In most cases we will want to test complex methods involving multiple users and money transfers. A perfect example for this is ourDonation Example , which enables users todonate money to a beneficiary. Lets see its integration tests

- JavaScript

contract-ts/sandbox-ts/src/main.ava.ts loading ... See full example on GitHub

Sandbox Testing

NEAR Workspaces allows you to write tests once, and run them either ontestnet or a localSandbox . Bydefault , Workspaces will start asandbox and run your testslocally . Lets dive into the features of our framework and see how they can help you.

Spooning Contracts

Spooning a blockchain is copying the data from one network into a different network. NEAR Workspaces makes it easy to copy data from Mainnet or Testnet contracts into your local Sandbox environment:

- JavaScript
- Rust

const refFinance =

await root . importContract ({ mainnetContract :

'v2.ref-finance.near' , blockId :

50_000_000 , withData :

true , }) ; This would copy the Wasm bytes and contract state fromv2.ref-finance.near to your local blockchain as it existed at block50_000_000 . This makes use of Sandbox's specialpatch state feature to keep the contract name the same, even though the top level account might not exist locally (note that this means it only works in Sandbox testing mode). You can then interact with the contract in a deterministic way the same way you interact with all other accounts created with near-workspaces.

note withData will only work out-of-the-box if the contract's data is 50kB or less. This is due to the default configuration of RPC servers; seethe "Heads Up" note here . See aTypeScript example of spooning contracts. Specify the contract name fromtestnet you want to be pulling, and a specific block ID referencing back to a specific time. (Just in case the contract you're referencing has been changed or updated)

const

CONTRACT_ACCOUNT :

& str

=

"contract_account_name_on_testnet.testnet" ; const

BLOCK_HEIGHT :

BlockHeight

=

12345 ; Create a function calledpull_contract which will pull the contract's.wasm file from the chain and deploy it onto your local sandbox. You'll have to re-initialize it with all the data to run tests.

async

fn

pull_contract (owner :

& Account , worker :

& Worker < Sandbox

)

->

anyhow :: Result < Contract

{ let testnet =

workspaces :: testnet_archival ( ) ; let contract_id :

AccountId

=

CONTRACT_ACCOUNT . parse ( ) ? ; This next line will actually pull down the relevant contract from testnet and set an initial balance on it with 1000 NEAR.

let contract = worker . import_contract (& contract_id ,

& testnet) . initial_balance (parse_near! ("1000 N")) . block_height (BLOCK_HEIGHT) . transact ( ) . await ? ; Following that you'll have to init the contract again with your metadata. This is because the contract's data is too big for the RPC service to pull down. (limits are set to 50Mb)

owner . call (& worker , contract . id ( ) ,

"init_method_name") . args_json (serde_json :: json! ({ "arg1" : value1 , "arg2" : value2 , })) ? . transact ( ) . await ? ; Ok (contract) }

Patch State on the Fly

In Sandbox-mode, you can add or modify any contract state, contract code, account or access key withpatchState .

tip You can alter contract code, accounts, and access keys using normal transactions via theDeployContract ,CreateAccount , andAddKey actions . But this limits you to altering your own account or sub-account.patchState allows you to perform these operations on any account. Keep in mind that you cannot perform arbitrary mutation on contract state with transactions since transactions can only include contract calls that mutate state in a contract-programmed way. For example, with an NFT contract, you can perform some operation with NFTs you have ownership of, but you cannot manipulate NFTs that are owned by other accounts since the smart contract is coded with checks to reject that. This is the expected behavior of the NFT contract. However, you may want to change another person's NFT for a test setup. This is called "arbitrary mutation on contract state" and can be done withpatchState :

- JavaScript
- Rust

const

{ contract , ali }

= t . context . accounts ; // Contract must have some state for viewState & patchState to work await ali . call (contract ,

'set_status' ,

{ message :

'hello' }) ; // Get state const state =

await contract . viewState ( ) ; // Get raw value const statusMessage = state . get ('STATE' ,

{ schema ,

type :

StatusMessage }) ; // Update contract state statusMessage . records . push (new

BorshRecord ({ k :

'alice.near' ,

v :

'hello world' }) ,) ; // Serialize and patch state back to runtime await contract . patchState ('STATE' , borsh . serialize (schema , statusMessage) ,) ; // Check again that the update worked const result =

await contract . view ('get_status' ,

{ account_id :

'alice.near' , }) ; t . is (result ,

'hello world') ; To see a complete example of how to do this, see the patch-state test . // Grab STATE from the testnet status_message contract. This contract contains the following data: // get_status(dev-20211013002148-59466083160385) => "hello from testnet" let

(testnet_contract_id , status_msg)

=

{ let worker =

workspaces :: testnet ( ) . await ? ; let contract_id :

AccountId

=

TESTNET_PREDEPLOYED_CONTRACT_ID . parse ( ) . map_err (anyhow :: Error :: msg) ? ; let

mut state_items = worker . view_state (& contract_id ,

None) . await ? ; let state = state_items . remove (b"STATE" . as_slice ( )) . unwrap ( ) ; let status_msg =

StatusMessage :: try_from_slice (& state) ? ; (contract_id , status_msg) } ; info! (target :

"spooning" ,

"Testnet: {:?}" , status_msg) ; // Create our sandboxed environment and grab a worker to do stuff in it: let worker =

workspaces :: sandbox ( ) . await ? ; // Deploy with the following status_message state: sandbox_contract_id => "hello from sandbox" let sandbox_contract =

deploy_status_contract (& worker ,

"hello from sandbox") . await ? ; // Patch our testnet STATE into our local sandbox: worker . patch_state (sandbox_contract . id ( ) , "STATE" . as_bytes ( ) , & status_msg . try_to_vec ( ) ? ,) . await ? ; // Now grab the state to see that it has indeed been patched: let status :

String

= sandbox_contract . view (& worker , "get_status" , serde_json :: json! ({ "account_id" : testnet_contract_id , }) . to_string ( ) . into_bytes ( ) ,) . await ? . json ( ) ? ; info! (target :

"spooning" ,

"New status patched: {:?}" , status) ; assert_eq! (& status ,

"hello from testnet") ; As an alternative topatchState , you can stop the node, dump state at genesis, edit the genesis, and restart the node. This approach is more complex to do and also cannot be performed without restarting the node.

Time Traveling

workspaces 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:

- JavaScript
- Rust

tests/08.fast-forward.ava.ts loading ... See full example on GitHub

[tokio::test]

async

fn

test_contract ( )

->

anyhow :: Result < ( )

{ let worker =

workspaces :: sandbox ( ) . await ? ; let contract = worker . dev_deploy (WASM_BYTES) ; let blocks_to_advance =

10000 ; worker . fast_forward (blocks_to_advance) ; // Now, "do_something_with_time" will be in the future and can act on future time-related state. contract . call (& worker ,

"do_something_with_time") . transact ( ) . await ? ; }See the full example on Github.

Using Testnet

NEAR Workspaces is set up so that you can write tests once and run them against a local Sandbox node (the default behavior) or againstNEAR TestNet . Some reasons this might be helpful:

- Gives higher confidence that your contracts work as expected
- You can test against deployed testnet contracts
- If something seems off in Sandbox mode, you can compare it to testnet

tip In order to use Workspaces in testnet mode you will need to have atestnet account. You can create onhere . You can switch to testnet mode in three ways.

1. When creating Worker set network totestnet
2. and pass your master account:
3. JavaScript
4. Rust

const worker =

await

Worker . init ({ network :

'testnet' , testnetMasterAccountId :

" , })

[tokio::main]

// or whatever runtime we want async

fn

main ( )

->

anyhow :: Result < ( )

{ // Create a sandboxed environment. // NOTE: Each call will create a new sandboxed environment let worker =

workspaces :: sandbox ( ) . await ? ; // or for testnet: let worker =

workspaces :: testnet ( ) . await ? ; } 1. Set theNEAR_WORKSPACES_NETWORK 2. andTESTNET_MASTER_ACCOUNT_ID 3. environment variables when running your tests:

- JavaScript

NEAR_WORKSPACES_NETWORK=testnet TESTNET_MASTER_ACCOUNT_ID= node test.js If you set this environment variables and pass{network: 'testnet', testnetMasterAccountId: } toWorker.init , the config object takes precedence. 1. If usingnear-workspaces 2. with AVA, you can use a custom config file. Other test runners allow similar config files; adjust the following instructions for your situation.

- JavaScript

Create a file in the same directory as yourpackage.json calledava.testnet.config.cjs with the following contents:

module . exports

=

{ ... require ('near-workspaces/ava.testnet.config.cjs') , ... require ('./ava.config.cjs') , } ; module . exports . environmentVariables

=

{ TESTNET_MASTER_ACCOUNT_ID :

" , } ; Thenear-workspaces/ava.testnet.config.cjs import sets theNEAR_WORKSPACES_NETWORK environment variable for you. A benefit of this approach is that you can then easily ignore files that should only run in Sandbox mode.

Now you'll also want to add atest:testnet script to yourpackage.json 'sscripts section:

"scripts": { "test": "ava", + "test:testnet": "ava --config ./ava.testnet.config.cjs" }

Additional Media

Test Driven Design Using Workspaces and AVA

The video below walks through how to apply TDD with Workspaces and AVA for a simple contractEdit this page Last updatedonApr 10, 2024 bygagdiez Was this page helpful? Yes No Need some help?Chat with us or check ourDev Resources ! Twitter Telegram Discord Zulip

Previous Unit Testing Next Local Development