Testing with NodeJS

When developing programs on Solana, ensuring their correctness and reliability is crucial. Until now devs have been usingsolana-test-validator for testing. This document covers testing your Solana program with Node.js usingsolana-bankrun .

Overview#

There are two ways to test programs on Solana:

1. solana-test-validator
2. :
3. That spins up a local emulator of the Solana Blockchain on your local machine
4. which receives the transactions to be processed by the validator.
5. The variousBanksClient-based
6. test frameworks for SBF (Solana Bytecode Format) programs: Bankrun is a
7. framework that simulates a Solana bank's operations, enabling developers to
8. deploy, interact with, and assess the behavior of programs under test
9. conditions that mimic the mainnet. It helps set up the test environment and
10. offers tools for detailed transaction insights, enhancing debugging and
11. verification. With the client, we can load programs, and simulate and process
12. transactions seamlessly.solana-program-test
13. (Rust),solana-bankrun
14. (Rust,
15. JavaScript),anchor-bankrun
16. (Anchor, JavaScript),solders.bankrun
17. (Python) are examples of the BanksClient-based testing framework.

Info pnpm create solana-program can help you generate JS and Rust clients including tests. Anchor is not yet supported. In this guide, we are using Solana Bankrun.Bankrun is a superfast, powerful, and lightweight framework for testing Solana programs in Node.js.

- The biggest advantage of using Solana Bankrun is that you don't have to set
- up
- an environment to test programs like you'd have to do while using the
- solana-test-validator
- . Instead, you can do that with a piece of code,
- inside
- the tests.
- It also dynamically sets time and account data, which isn't possible with
- solana-test-validator

Installation#

Addsolana-bankrun as a dev dependency to your node project. If your Solana program is not a node project yet, you can initialize it usingnpm init .

npm i -D solana-bankrun

Usage#

Program Directory#

Firstly, the program's.so file must be present in one of the following directories:

- ./tests/fixtures
- (just create this directory if it doesn't exist already).
- Your current working directory.
- A directory you define in theBPF_OUT_DIR
- orSBF_OUT_DIR
- environment
- variables.export BPF_OUT_DIR='/path/to/binary'
- Build your program specifying the correct directory so that library can pick
- the file up from directory just from the name.cargo build-sbf --manifest-path=./program/Cargo.toml --sbf-out-dir=./tests/fixtures

Testing Framework#

solana-bankrun is used in JavaScript or TypeScript with testing frameworks likets-mocha ,ava ,Jest , etc. Make sure to get started with any of the above.

Add annpm script to test your program and create yourtest.ts file insidetests folder.

{ "scripts" : { "test" : "pnpm ts-mocha -p ./tsconfig.json -t 1000000 ./tests/test.ts" } }

Start#

start function fromsolana-bankrun spins up a BanksServer and a BanksClient, deploy programs and add accounts as instructed.

import { start } from "solana-bankrun" ; import { PublicKey } from "@solana/web3.js" ;

test ("testing program instruction" , async () => { const programId = PublicKey. unique (); const context = await start ([{ name: "program_name" , programId }], []);

const client = context.banksClient; const payer = context.payer; // write tests });

Bankruncontext

#

- We get access to the Bankruncontext
- from thestart
- function. Context
- contains a BanksClient, a recent blockhash and a funded payer keypair.
- context
- has apayer
- , which is a funded keypair that can be used to sign
- transactions.
- context
- also hascontext.lastBlockhash
- orcontext.getLatestBlockhash
- to
- make fetchingBlockhash
- convenient during tests.
- context.banksClient
- is used to send transactions and query account data from
- the ledger state. For example, sometimesRent
- (in lamports) is
- required to build a transaction to be submitted, for example, when using the
- SystemProgram's
- createAccount() instruction. You can do that using BanksClient:
- const
- rent
- =
- await
- client.
- getRent
- ();
- const
- Ix
- :
- TransactionInstruction
- =
- SystemProgram.
- createAccount
- ({
- // ...
- lamports:
- Number
- (rent.
- minimumBalance
- (
- BigInt
- (

- ACCOUNT_SIZE
-))),
- //....
- });
- You can read account data from BanksClient usinggetAccount
- function
- AccountInfo
- =
- await
- client.
- getAccount
- (counter);

Process Transaction#

TheprocessTransaction() function executes the transaction with the loaded programs and accounts from the start function and will return a transaction.

let transaction = await client. processTransaction (tx);

Example#

Here's an example to write test for ahello world program :

import { PublicKey, Transaction, TransactionInstruction, } from "@solana/web3.js" ; import { start } from "solana-bankrun" ; import { describe, test } from "node:test" ; import { assert } from "chai" ;

describe ("hello-solana" , async () => { // load program in solana-bankrun const PROGRAM_ID = PublicKey. unique (); const context = await start ([{ name: "hello_solana_program" , programId: PROGRAM_ID }], [],); const client = context.banksClient; const payer = context.payer;

test ("Say hello!" , async () => { const blockhash = context.lastBlockhash; // We set up our instruction first. let ix = new TransactionInstruction ({ // using payer keypair from context to sign the txn keys: [{ pubkey: payer.publicKey, isSigner: true , isWritable: true }], programId: PROGRAM_ID , data: Buffer. alloc (0), // No data });

const tx = new Transaction (); tx.recentBlockhash = blockhash; // using payer keypair from context to sign the txn tx. add (ix). sign (payer);

// Now we process the transaction let transaction = await client. processTransaction (tx);

assert (transaction.logMessages[0]. startsWith ("Program " + PROGRAM_ID)); assert (transaction.logMessages[1] === "Program log: Hello, Solana!"); assert (transaction.logMessages[2] === "Program log: Our program's Program ID: " + PROGRAM_ID ,); assert (transaction.logMessages[3]. startsWith ("Program " + PROGRAM_ID + " consumed" ,),); assert (transaction.logMessages[4] === "Program " + PROGRAM_ID + " success"); assert (transaction.logMessages. length == 5); }); }); This is how the output looks like after running the tests forhello world program .

[2024-06-04T12:57:36.188822000Z INFO solana_program_test] "hello_solana_program" SBF program from tests/fixtures/hello_solana_program.so, modified 3 seconds, 20 ms, 687 µs and 246 ns ago [2024-06-04T12:57:36.246838000Z DEBUG solana_runtime::message_processor::stable_log] Program 11111111111111111111111111111112 invoke [1] [2024-06-04T12:57:36.246892000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Hello, Solana! [2024-06-04T12:57:36.246917000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Our program's Program ID: 11111111111111111111111111111112 [2024-06-04T12:57:36.246932000Z DEBUG solana_runtime::message_processor::stable_log] Program 11111111111111111111111111111112 consumed 2905 of 200000 compute units [2024-06-04T12:57:36.246937000Z DEBUG solana_runtime::message_processor::stable_log] Program 11111111111111111111111111111112 success ▶ hello-solana ✔ Say hello! (5.667917ms) ▶ hello-solana (7.047667ms)

ℹ tests 1 **ℹ** suites 1 **ℹ** pass 1 **ℹ** fail 0 **ℹ** cancelled 0 **ℹ** skipped 0 **ℹ** todo 0 **ℹ** duration_ms 63.52616

Previous «Program Examples Next Limitations»