Developing Programs in Rust

Solana programs are primarily developed using the Rust programming language. This page focuses on writing Solana programs in Rust without using the Anchor framework, an approach often referred to as writing "native Rust" programs.

Native Rust development provides developers with direct control over their Solana programs. However, this approach requires more manual setup and boilerplate code compared to using the Anchor framework. This method is recommended for developers who:

- Seek granular control over program logic and optimizations
- Want to learn the underlying concepts before moving to higher-level frameworks

For beginners, we recommend starting with the Anchor framework. See theAnchor section for more information.

Prerequisites#

For detailed installation instructions, visit theinstallation page.

Before you begin, ensure you have the following installed:

- Rust: The programming language for building Solana programs.
- Solana CLI: Command-line tool for Solana development.

Getting Started#

The example below covers the basic steps to create your first Solana program written in Rust. We'll create a minimal program that prints "Hello, world!" to the program log.

Create a new Program#

First, create a new Rust project using the standardcargo init command with the--lib flag.

Terminal cargo init hello_world --lib Navigate to the project directory. You should see the defaultsrc/lib.rs andCargo.toml files

Terminal cd hello_world Next, add thesolana-program dependency. This is the minimum dependency required to build a Solana program.

Terminal cargo add solana-program@1.18.26 Next, add the following snippet toCargo.toml . If you don't include this config, thetarget/deploy directory will not be generated when you build the program.

Cargo.toml [lib] crate-type = ["cdylib" , "lib"] YourCargo.toml file should look like the following:

Cargo.toml [package] name = "hello_world" version = "0.1.0" edition = "2021"

[lib] crate-type = ["cdylib" , "lib"]

[dependencies] solana-program = "1.18.26" Next, replace the contents ofsrc/lib.rs with the following code. This is a minimal Solana program that prints "Hello, world!" to the program log when the program is invoked.

Themsg! macro is used in Solana programs to print a message to the program log.

lib.rs use solana_program :: { account_info :: AccountInfo , entrypoint, entrypoint :: ProgramResult , msg, pubkey :: Pubkey , };

entrypoint! (process_instruction);

pub fn process_instruction (_program_id : & Pubkey , _accounts : & [AccountInfo], _instruction_data : & [u8],) -> ProgramResult { msg! ("Hello, world!"); Ok (()) }

Build the Program#

Next, build the program using thecargo build-sbf command.

Terminal cargo build-sbf This command generates atarget/deploy directory containing two important files:

1. A.so
2. file (e.g.,hello_world.so
3.): This is the compiled Solana program
4. that will be deployed to the network as a "smart contract".

5. A keypair file (e.g.,hello_world-keypair.json
6.): The public key of this
7. keypair is used as the program ID when deploying the program.

To view the program ID, run the following command in your terminal. This command prints the public key of the keypair at the specified file path:

Terminal solana address -k ./target/deploy/hello_world-keypair.json Example output:

4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz

Test the Program#

Next, test the program using thesolana-program-test crate. Add the following dependencies toCargo.toml .

Terminal cargo add solana-program-test@1.18.26 --dev cargo add solana-sdk@1.18.26 --dev cargo add tokio --dev Add the following test tosrc/lib.rs , below the program code. This is a test module that invokes the hello world program.

lib.rs

[cfg(test)]

mod test { use super :: *; use solana_program_test ::*; use solana_sdk :: { signature :: Signer , transaction :: Transaction };

[tokio

:: test] async fn test_hello_world () { let program_id = Pubkey :: new_unique (); let (mut banks_client, payer, recent_blockhash) = ProgramTest :: new ("hello_world" , program_id, processor! (process_instruction)) . start () .await ;

// Create the instruction to invoke the program let instruction = solana_program :: instruction :: Instruction :: new_with_borsh (program_id, & (), vec! []);

// Add the instruction to a new transaction let mut transaction = Transaction :: new_with_payer (& [instruction], Some (& payer . pubkey ())); transaction . sign (& [& payer], recent_blockhash);

// Process the transaction let transaction_result = banks_client . process_transaction (transaction) .await ; assert! (transaction_result . is_ok ()); } } Run the test using thecargo test-sbf command. The program log will display "Hello, world!".

Terminal cargo test-sbf Example output:

Terminal running 1 test [2024-10-18T21:24:54.889570000Z INFO solana_program_test] "hello_world" SBF program from /hello_world/target/deploy/hello_world.so, modified 35 seconds, 828 ms, 268 µs and 398 ns ago [2024-10-18T21:24:54.974294000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM invoke [1] [2024-10-18T21:24:54.974814000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Hello, world ! [2024-10-18T21:24:54.976848000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM consumed 140 of 200000 compute units [2024-10-18T21:24:54.976868000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM success test test::test_hello_world ... ok

test result: ok. 1 passed ; 0 failed ; 0 ignored ; 0 measured ; 0 filtered out ; finished in 0.13s

Deploy the Program#

Next, deploy the program. When developing locally, we can use thesolana-test-validator .

First, configure the Solana CLI to use the local Solana cluster.

Terminal solana config set -ul Example output:

Config File: /.config/solana/cli/config.yml RPC URL: http://localhost:8899 WebSocket URL: ws://localhost:8900/ (computed) Keypair Path: /.config/solana/id.json Commitment: confirmed Open a new terminal and run thesolana-test-validators command to start the local validator.

Terminal solana-test-validator While the test validator is running, run thesolana program deploy command in a separate terminal to deploy the program to the local validator.

Terminal solana program deploy ./target/deploy/hello_world.so Example output:

Program Id: 4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz Signature: 5osMiNMiDZGM7L1e2tPHxU8wdB8gwG8fDnXLg5G7SbhwFz4dHshYgAijk4wSQL5cXiu8z1MMou5kLadAQuHp7ybH You can inspect the program ID and transaction signature onSolana Explorer . Note that the cluster on Solana Explorer must also be localhost. The "Custom RPC URL" option on Solana Explorer defaults tohttp://localhost:8899 .

Invoke the Program#

Next, we'll demonstrate how to invoke the program using a Rust client.

First create anexamples directory and aclient.rs file.

Terminal mkdir -p examples touch examples/client.rs Add the following toCargo.toml .

Cargo.toml [[example]] name = "client" path = "examples/client.rs" Add thesolana-client dependency.

Terminal cargo add solana-client@1.18.26 --dev Add the following code toexamples/client.rs . This is a Rust client script that funds a new keypair to pay for transaction fees and then invokes the hello world program.

example/client.rs use solana_client :: rpc_client :: RpcClient ; use solana_sdk :: { commitment_config :: CommitmentConfig , instruction :: Instruction , pubkey :: Pubkey , signature :: { Keypair , Signer }, transaction :: Transaction , }; use std :: str :: FromStr ;

[tokio

:: main] async fn main () { // Program ID (replace with your actual program ID) let program_id = Pubkey :: from_str ("4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz") . unwrap ();

// Connect to the Solana devnet let rpc_url = String :: from ("http://127.0.0.1:8899"); let client = RpcClient :: new_with_commitment (rpc_url, CommitmentConfig :: confirmed ());

// Generate a new keypair for the payer let payer = Keypair :: new ();

// Request airdrop let airdrop_amount = 1_000_000_000 ; // 1 SOL let signature = client . request_airdrop (& payer . pubkey (), airdrop_amount) . expect ("Failed to request airdrop");

// Wait for airdrop confirmation loop { let confirmed = client . confirm_transaction (& signature) . unwrap (); if confirmed { break ; } }

// Create the instruction let instruction = Instruction :: new_with_borsh (program_id, & (), // Empty instruction data vec! [], // No accounts needed);

// Add the instruction to new transaction let mut transaction = Transaction :: new_with_payer (& [instruction], Some (& payer . pubkey ())); transaction . sign (& [& payer], client . get_latest_blockhash () . unwrap ());

// Send and confirm the transaction match client . send_and_confirm_transaction (& transaction) { Ok (signature) => println! ("Transaction Signature: {}" , signature), Err (err) => eprintln! ("Error sending transaction: {}" , err), } } Before running the script, replace the program ID in the code snippet above with the one for your program.

You can get your program ID by running the following command.

Terminal solana address -k ./target/deploy/hello_world-keypair.json

[tokio::main]

async fn main() { - let program_id = Pubkey::from_str("4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz").unwrap(); + let program_id = Pubkey::from_str("YOUR_PROGRAM_ID).unwrap(); } } Run the client script with the following command.

Terminal cargo run --example client Example output:

Transaction Signature: 54TWxKi3Jsi3UTeZbhLGUFX6JQH7TspRJjRRFZ8NFnwG5BXM9udxiX77bAACjKAS9fGnVeEazrXL4SfKrW7xZFYV You can inspect the transaction signature onSolana Explorer (local cluster) to see "Hello, world!" in the program log.

Update the Program#

Solana programs can be updated by redeploying to the same program ID. Update the program insrc/lib.rs to print "Hello, Solana!" instead of "Hello, world!".

lib.rs pub fn process_instruction(_program_id: &Pubkey, _accounts: &[AccountInfo], _instruction_data: &[u8],) -> ProgramResult { - msg!("Hello, world!"); + msg!("Hello, Solana!"); Ok(()) } Test the updated program by running thecargo test-sbf command.

Terminal cargo test-sbf You should see "Hello, Solana!" in the program log.

Terminal running 1 test [2024-10-23T19:28:28.842639000Z INFO solana_program_test] "hello_world" SBF program from /code/misc/delete/hello_world/target/deploy/hello_world.so, modified 4 minutes, 31 seconds, 435 ms, 566 µs and 766 ns ago [2024-10-23T19:28:28.934854000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM invoke [1] [2024-10-23T19:28:28.936735000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Hello, Solana ! [2024-10-23T19:28:28.938774000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM consumed 140 of 200000 compute units [2024-10-23T19:28:28.938793000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM success test test::test_hello_world ... ok

test result: ok. 1 passed ; 0 failed ; 0 ignored ; 0 measured ; 0 filtered out ; finished in 0.14s Run thecargo build-sbf command to generate an updated.so file.

Terminal cargo build-sbf Redeploy the program using thesolana program deploy command.

Terminal solana program deploy ./target/deploy/hello_world.so Run the client code again and inspect the transaction signature on Solana Explorer to see "Hello, Solana!" in the program log.

Terminal cargo run --example client

Close the Program#

You can close your Solana program to reclaim the SOL allocated to the account. Closing a program is irreversible, so it should be done with caution.

To close a program, use thesolana program closecommand. For example:

Terminal solana program close 4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz --bypass-warning Example output:

Closed Program Id 4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz, 0.1350588 SOL reclaimed Note that once a program is closed, its program ID cannot be reused. Attempting to deploy a program with a previously closed program ID will result in an error.

Error: Program 4Ujf5fXfLx2PAwRqcECCLtgDxHKPznoJpa43jUBxFfMz has been closed, use a new Program Id If you need to redeploy a program with the same source code after closing a program, you must generate a new program ID. To generate a new keypair for the program, run the following command:

Terminal solana-keygen new -o ./target/deploy/hello_world-keypair.json --force Alternatively, you can delete the existing keypair file (e.g../target/deploy/hello_world-keypair.json) and runcargo build-sbf again, which will generate a new keypair file.

Previous «CPIs with Anchor Next Program Structure»