Rust Program Structure

Solana programs written in Rust have minimal structural requirements, allowing for flexibility in how code is organized. The only requirement is that a program must have anentrypoint , which defines where the execution of a program begins.

Program Structure#

While there are no strict rules for file structure, Solana programs typically follow a common pattern:

- entrypoint.rs
- : Defines the entrypoint that routes incoming instructions.
- state.rs
- : Define program-specific state (account data).
- instructions.rs
- : Defines the instructions that the program can execute.
- processor.rs
- : Defines the instruction handlers (functions) that implement
- the business logic for each instruction.
- error.rs
- : Defines custom errors that the program can return.

You can find examples in theSolana Program Library .

Example Program#

To demonstrate how to build a native Rust program with multiple instructions, we'll walk through a simple counter program that implements two instructions:

1. InitializeCounter
2. : Creates and initializes a new account with an initial
3. value.
4. IncrementCounter
5. : Increments the value stored in an existing account.

For simplicity, the program will be implemented in a singlelib.rs file, though in practice you may want to split larger programs into multiple files.

Full Program Code

Create a new Program#

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

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

Terminal cd counter_program 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 = "counter_program" version = "0.1.0" edition = "2021"

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

[dependencies] solana-program = "1.18.26"

Program Entrypoint#

A Solana program entrypoint is the function that gets called when a program is invoked. The entrypoint has the following raw definition and developers are free to create their own implementation of the entrypoint function.

For simplicity, use theentrypoint! macro from thesolana_program crate to define the entrypoint in your program.

[no_mangle]

pub unsafe extern "C" fn entrypoint (input : *mut u8) -> u64 ; Replace the default code inlib.rs with the following code. This snippet:

1. Imports the required dependencies fromsolana_program
2. Defines the program entrypoint using theentrypoint!
3. macro
4. Implements theprocess_instruction
5. function that will route instructions to
6. the appropriate handler functions

lib.rs use solana_program :: { account_info :: {next_account_info, AccountInfo }, entrypoint, entrypoint :: ProgramResult , msg, program :: invoke, program_error :: ProgramError , pubkey :: Pubkey , system_instruction, sysvar :: { rent :: Rent , Sysvar }, };

entrypoint! (process_instruction);

pub fn

process_instruction (program_id : & Pubkey , accounts : & [AccountInfo], instruction_data : & [u8],) -> ProgramResult { // Your program logic Ok (()) } Theentrypoint! macro requires a function with the the followingtype signature as an argument:

pub type ProcessInstruction = fn (program_id : & Pubkey , accounts : & [AccountInfo], instruction_data : & [u8]) -> ProgramResult ; When a Solana program is invoked, the entrypointdeserializes theinput data (provided as bytes) into three values and passes them to theprocess_instruction function:

- program_id
- : The public key of the program being invoked (current program)
- accounts
- : TheAccountInfo
- for accounts required by the instruction being
- invoked
- instruction_data
- : Additional data passed to the program which specifies the
- instruction to execute and its required arguments

These three parameters directly correspond to the data that clients must provide when building an instruction to invoke a program.

Define Program State#

When building a Solana program, you'll typically start by defining your program's state - the data that will be stored in accounts created and owned by your program.

Program state is defined using Rust structs that represent the data layout of your program's accounts. You can define multiple structs to represent different types of accounts for your program.

When working with accounts, you need a way to convert your program's data types to and from the raw bytes stored in an account's data field:

- Serialization: Converting your data types into bytes to store in an account's
- data field
- Deserialization: Converting the bytes stored in an account back into your data
- types

While you can use any serialization format for Solana program development,Borsh is commonly used. To use Borsh in your Solana program:

1. Add theborsh
2. crate as a dependency to yourCargo.toml
3. :

Terminal cargo add borsh 1. Import the Borsh traits and use the derive macro to implement the traits for 2. your structs:

use borsh :: { BorshSerialize , BorshDeserialize };

// Define struct representing our counter account's data

[derive(

BorshSerialize , BorshDeserialize , Debug)] pub struct CounterAccount { count : u64 , } Add theCounterAccount struct tolib.rs to define the program state. This struct will be used in both the initialization and increment instructions.

lib.rs use solana_program :: { account_info :: {next_account_info, AccountInfo }, entrypoint, entrypoint :: ProgramResult , msg, program :: invoke, program_error :: ProgramError , pubkey :: Pubkey , system_instruction, sysvar :: { rent :: Rent , Sysvar }, }; use borsh :: { BorshSerialize , BorshDeserialize };

entrypoint! (process_instruction);

pub fn process_instruction (program_id : & Pubkey , accounts : & [AccountInfo], instruction_data : & [u8],) -> ProgramResult { // Your program logic Ok (()) }

[derive(

BorshSerialize , BorshDeserialize , Debug)] pub struct CounterAccount { count : u64 , }

Define Instructions#

Instructions refer to the different operations that your Solana program can perform. Think of them as public APIs for your program - they define what actions users can take when interacting with your program.

Instructions are typically defined using a Rust enum where:

- Each enum variant represents a different instruction
- The variant's payload represents the instruction's parameters

Note that Rust enum variants are implicitly numbered starting from 0.

Below is an example of an enum defining two instructions:

[derive(

BorshSerialize , BorshDeserialize , Debug)] pub enum CounterInstruction { InitializeCounter { initial_value : u64 }, // variant 0 IncrementCounter , // variant 1 } When a client invokes your program, they must provide instruction data (as a buffer of bytes) where:

- The first byte identifies which instruction variant to execute (0, 1, etc.)
- The remaining bytes contain the serialized instruction parameters (if
- required)

To convert the instruction data (bytes) into a variant of the enum, it is common to implement a helper method. This method:

1. Splits the first byte to get the instruction variant
2. Matches on the variant and parses any additional parameters from the
3. remaining bytes
4. Returns the corresponding enum variant

For example, theunpack method for theCounterInstruction enum:

impl CounterInstruction { pub fn unpack (input : & [u8]) -> Result < Self , ProgramError

> { // Get the instruction variant from the first byte let (& variant, rest) = input . split_first () . ok_or (ProgramError :: InvalidInstructionData) ? ;

// Match instruction type and parse the remaining bytes based on the variant match variant { 0 => { // For InitializeCounter, parse a u64 from the remaining bytes let initial_value = u64 :: from_le_bytes (rest . try_into () . map_err (| _ | ProgramError :: InvalidInstructionData) ?); Ok (Self :: InitializeCounter { initial_value }) } 1 => Ok (Self :: IncrementCounter), // No additional data needed _ => Err (ProgramError :: InvalidInstructionData), } } } Add the following code tolib.rs to define the instructions for the counter program.

lib.rs use borsh :: { BorshDeserialize , BorshSerialize }; use solana_program :: { account_info :: AccountInfo , entrypoint, entrypoint :: ProgramResult , msg, program_error :: ProgramError , pubkey :: Pubkey , };

entrypoint! (process_instruction);

pub fn process_instruction (program_id : & Pubkey , accounts : & [AccountInfo], instruction_data : & [u8],) -> ProgramResult { // Your program logic Ok (()) }

[derive(

BorshSerialize , BorshDeserialize , Debug)] pub enum CounterInstruction { InitializeCounter { initial_value : u64 }, // variant 0 IncrementCounter , // variant 1 }

impl CounterInstruction { pub fn unpack (input : & [u8]) -> Result < Self , ProgramError

> { // Get the instruction variant from the first byte let (& variant, rest) = input . split_first () . ok_or (ProgramError :: InvalidInstructionData) ? ;

// Match instruction type and parse the remaining bytes based on the variant match variant { 0 => { // For InitializeCounter, parse a u64 from the remaining bytes let initial_value = u64 :: from_le_bytes (rest . try_into () . map_err (| _ | ProgramError :: InvalidInstructionData) ? ,); Ok (Self :: InitializeCounter { initial_value }) } 1 => Ok (Self :: IncrementCounter), // No additional data needed _ => Err (ProgramError :: InvalidInstructionData), } } }

Instruction Handlers#

Instruction handlers refer to the functions that contain the business logic for each instruction. It's common to name handler functions asprocess_ , but you're free to choose any naming convention.

Add the following code tolib.rs . This code uses theCounterInstruction enum andunpack method defined in the previous step to route incoming instructions to the appropriate handler functions:

lib.rs entrypoint! (process_instruction);

pub fn process_instruction (program_id : & Pubkey , accounts : & [AccountInfo], instruction_data : & [u8],) -> ProgramResult { // Unpack instruction data let instruction = CounterInstruction :: unpack (instruction_data) ? ;

// Match instruction type match instruction { CounterInstruction :: InitializeCounter { initial_value } => { process_initialize_counter (program_id, accounts, initial_value) ? } CounterInstruction :: IncrementCounter =>

process_increment_counter (program_id, accounts) ? , }; }

fn process_initialize_counter (program_id : & Pubkey , accounts : & [AccountInfo], initial_value : u64 ,) -> ProgramResult { // Implementation details... Ok (()) }

fn process_increment_counter (program_id : & Pubkey , accounts : & [AccountInfo]) -> ProgramResult { // Implementation details... Ok (()) } Next, add the implementation of theprocess_initialize_counter function. This instruction handler:

1. Creates and allocates space for a new account to store the counter data
2. Initializing the account data withinitial_value
3. passed to the instruction

Explanation

lib.rs // Initialize a new counter account fn process_initialize_counter (program_id : & Pubkey , accounts : & [AccountInfo], initial_value : u64 ,) -> ProgramResult { let accounts_iter = &mut accounts . iter ();

let counter_account = next_account_info (accounts_iter) ? ; let payer_account = next_account_info (accounts_iter) ? ; let system_program = next_account_info (accounts_iter) ? ;

// Size of our counter account let account_space = 8 ; // Size in bytes to store a u64

// Calculate minimum balance for rent exemption let rent = Rent :: get () ? ; let required_lamports = rent . minimum_balance (account_space);

// Create the counter account invoke (& system_instruction :: create_account (payer_account . key, // Account paying for the new account counter_account . key, // Account to be created required_lamports, // Amount of lamports to transfer to the new account account_space as u64 , // Size in bytes to allocate for the data field program_id, // Set program owner to our program), & [payer_account . clone (), counter_account . clone (), system_program . clone (),],) ? ;

// Create a new CounterAccount struct with the initial value let counter_data = CounterAccount { count : initial_value, };

// Get a mutable reference to the counter account's data let mut account_data = &mut counter_account . data . borrow_mut ()[..];

// Serialize the CounterAccount struct into the account's data counter_data . serialize (&mut account_data) ? ;

msg! ("Counter initialized with value: {}" , initial_value);

Ok (()) } Next, add the implementation of theprocess_increment_counter function. This instruction increments the value of an existing counter account.

Explanation

lib.rs // Update an existing counter's value fn process_increment_counter (program_id : & Pubkey , accounts : & [AccountInfo]) -> ProgramResult { let accounts_iter = &mut accounts . iter (); let counter_account = next_account_info (accounts_iter) ? ;

// Verify account ownership if counter_account . owner != program_id { return Err (ProgramError :: IncorrectProgramId); }

// Mutable borrow the account data let mut data = counter_account . data . borrow_mut ();

// Deserialize the account data into our CounterAccount struct let mut counter_data : CounterAccount = CounterAccount :: try_from_slice (& data) ? ;

// Increment the counter value counter_data . count = counter_data . count . checked_add (1) . ok_or (ProgramError :: InvalidAccountData) ? ;

// Serialize the updated counter data back into the account counter_data . serialize (&mut &mut data[..]) ? ;

msg! ("Counter incremented to: {}" , counter_data . count); Ok (()) }

Instruction Testing[#]

To test the program instructions, 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 Then add the following test module tolib.rs and runcargo test-sbf to execute the tests. Optionally, use the--nocapture flag to see the print statements in the output.

Terminal cargo test-sbf -- --nocapture

Explanation

lib.rs

[cfg(test)]

mod test { use super :: *; use solana_program_test :: *; use solana_sdk :: { instruction :: { AccountMeta , Instruction }, signature :: { Keypair , Signer }, system_program, transaction :: Transaction , };

[tokio

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

// Create a new keypair to use as the address for our counter account let counter_keypair = Keypair :: new (); let initial_value : u64 = 42 ;

// Step 1: Initialize the counter println! ("Testing counter initialization...");

// Create initialization instruction let mut init_instruction_data = vec! [0]; // 0 = initialize instruction init_instruction_data . extend_from_slice (& initial_value . to_le_bytes ());

let initialize_instruction = Instruction :: new_with_bytes (program_id, & init_instruction_data, vec! [AccountMeta :: new (counter_keypair . pubkey (), true), AccountMeta :: new (payer . pubkey (), true), AccountMeta :: new_readonly (system_program :: id (), false),],);

// Send transaction with initialize instruction let mut transaction = Transaction :: new_with_payer (& [initialize_instruction], Some (& payer . pubkey ())); transaction . sign (& [& payer, & counter_keypair], recent_blockhash); banks_client . process_transaction (transaction) .await. unwrap ();

// Check account data let account = banks_client . get_account (counter_keypair . pubkey ()) .await . expect ("Failed to get counter account");

if let Some (account_data) = account { let counter : CounterAccount = CounterAccount :: try_from_slice (& account_data . data) . expect ("Failed to deserialize counter data"); assert_eq! (counter . count, 42); println! ("✅ Counter initialized successfully with value: {}" , counter . count); }

// Step 2: Increment the counter println! ("Testing counter increment...");

// Create increment instruction let increment_instruction = Instruction :: new_with_bytes (program_id, & [1], // 1 = increment instruction vec! [AccountMeta :: new (counter_keypair . pubkey (), true)],);

// Send transaction with increment instruction let mut transaction = Transaction :: new_with_payer (& [increment_instruction], Some (& payer . pubkey ())); transaction . sign (& [& payer, & counter_keypair], recent_blockhash); banks_client . process_transaction (transaction) .await. unwrap ();

// Check account data let account = banks_client . get_account (counter_keypair . pubkey ()) .await . expect ("Failed to get counter account");

if let Some (account_data) = account { let counter : CounterAccount = CounterAccount :: try_from_slice (& account_data . data) . expect ("Failed to deserialize counter data"); assert_eq! (counter . count, 43); println! ("✅ Counter incremented successfully to: {}" , counter . count); } } } Example output:

Terminal running 1 test [2024-10-29T20:51:13.783708000Z INFO solana_program_test] "counter_program" SBF program from /counter_program/target/deploy/counter_program.so, modified 2 seconds, 169 ms, 153 µs and 461 ns ago [2024-10-29T20:51:13.855204000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM invoke [1] [2024-10-29T20:51:13.856052000Z DEBUG solana_runtime::message_processor::stable_log] Program 11111111111111111111111111111111 invoke [2] [2024-10-29T20:51:13.856135000Z DEBUG solana_runtime::message_processor::stable_log] Program 11111111111111111111111111111111 success [2024-10-29T20:51:13.856242000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Counter initialized with value: 42 [2024-10-29T20:51:13.856285000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM consumed 3791 of 200000 compute units [2024-10-29T20:51:13.856307000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM success [2024-10-29T20:51:13.860038000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM invoke [1] [2024-10-29T20:51:13.860333000Z DEBUG solana_runtime::message_processor::stable_log] Program log: Counter incremented to: 43 [2024-10-29T20:51:13.860355000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM consumed 756 of 200000 compute units [2024-10-29T20:51:13.860375000Z DEBUG solana_runtime::message_processor::stable_log] Program 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM success test test::test_counter_program ... ok

test result: ok. 1 passed ; 0 failed ; 0 ignored ; 0 measured ; 0 filtered out ; finished in 0.08s

[Previous «Rust Programs]() [Next Deploying Programs»]()