CPIs with Anchor

Cross Program Invocations (CPI) refer to the process of one program invoking instructions of another program, which enables the composibility of programs on Solana.

This section will cover the basics of implementing CPIs in an Anchor program, using a simple SOL transfer instruction as a practical example. Once you understand the basics of how to implement a CPI, you can apply the same concepts for any instruction.

Cross Program Invocations#

Let's examine a program that implements a CPI to the System Program's transfer instruction. Here is the example program onSolana Playground .

Thelib.rs file includes a singlesol_transfer instruction. When thesol_transfer instruction on the Anchor program is invoked, the program internally invokes the transfer instruction of the System Program.

lib.rs use anchor_lang :: prelude ::* ; use anchor_lang :: system_program :: { transfer , Transfer };

declare_id! ("9AvUNHjxscdkiKQ8tUn12QCMXtcnbR9BVGq3ULNzFMRi");

[program]

pub mod cpi { use super ::* ;

pub fn

sol_transfer (ctx : Context < SolTransfer

> , amount : u64) -> Result <()> { let from_pubkey = ctx . accounts . sender . to_account_info (); let to_pubkey = ctx . accounts . recipient . to_account_info (); let program_id = ctx . accounts . system_program . to_account_info ();

let cpi_context = CpiContext :: new (program_id, Transfer { from : from_pubkey, to : to_pubkey, },);

transfer (cpi_context, amount) ? ; Ok (()) } }

[derive(

Accounts)] pub struct SolTransfer <' info

> {

[account(

mut)] sender : Signer <' info

> ,

[account(

mut)] recipient : SystemAccount <' info

> , system_program : Program <' info , System , } Thecpi.test.ts file shows how to invoke the Anchor program'ssol_transfer instruction and logs a link to the transaction details on SolanaFM.

cpi.test.ts it ("SOL Transfer Anchor" , async () => { const transactionSignature = await program.methods . solTransfer (new BN (transferAmount)) . accounts ({ sender: sender.publicKey, recipient: recipient.publicKey, }) . rpc ();

console. log (\n Transaction Signature: + https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); }); You can build, deploy, and run the test for this example on Playground to view the transaction details on theSolanaFM explorer .

The transaction details will show that the Anchor program was first invoked (instruction 1), which then invokes the System Program (instruction 1.1), resulting in a successful SOL transfer.

Transaction Details

Example 1 Explanation#

Implementing a CPI follows the same pattern as building an instruction to add to a transaction. When implementing a CPI, we must specify the program ID, accounts, and instruction data for the instruction being called.

The System Program's transfer instruction requires two accounts:

- from
- : The account sending SOL.
- to
- : The account receiving SOL.

In the example program, theSolTransfer struct specifies the accounts required by the transfer instruction. The System Program is also included because the CPI invokes the System Program.

[derive(

Accounts)] pub struct SolTransfer <' info

> {

[account(

mut)] sender : Signer <' info

> , // from account

[account(

mut)] recipient : SystemAccount <' info

> , // to account system_program : Program <' info , System , // program ID } The following tabs present three approaches to implementing Cross Program Invocations (CPIs), each at a different level of abstraction. All examples are functionally equivalent. The main purpose is to illustrate the implementation details of the CPI.

1 2 3 Thesol_transfer instruction included in the example code shows a typical approach for constructing CPIs using the Anchor framework.

This approach involves creating aCpiContext , which includes theprogram_id and accounts required for the instruction being called, followed by a helper function (transfer) to invoke a specific instruction.

use anchor_lang :: system_program :: {transfer, Transfer }; pub fn sol_transfer (ctx : Context < SolTransfer

> , amount : u64) -> Result <()> { let from_pubkey = ctx . accounts . sender . to_account_info (); let to_pubkey = ctx . accounts . recipient . to_account_info (); let program_id = ctx . accounts . system_program . to_account_info ();

let

cpi_context

= CpiContext :: new (program_id, Transfer { from : from_pubkey, to : to_pubkey, },);

transfer (cpi_context , amount) ? ; Ok (()) } Thecpi_context variable specifies the program ID (System Program) and accounts (sender and recipient) required by the transfer instruction.

let cpi_context = CpiContext :: new (program_id , Transfer { from :

from_pubkey , to :

to_pubkey , },); Thecpi_context andamount are then passed into thetransfer function to execute the CPI invoking the transfer instruction of the System Program.

transfer (cpi_context, amount) ? ; Here is a reference program onSolana Playground which includes all 3 examples.

Cross Program Invocations with PDA Signers#

Next, let's examine a program that implements a CPI to the System Program's transfer instruction where the sender is a Program Derived Address (PDA) that must be "signed" for by the program. Here is the example program onSolana Playground .

Thelib.rs file includes the following program with a singlesol_transfer instruction.

lib.rs use anchor_lang :: prelude ::* ; use anchor_lang :: system_program :: {transfer, Transfer };

declare_id! ("3455LkCS85a4aYmSeNbRrJsduNQfYRY82A7eCD3yQfyR");

[program]

pub mod cpi { use super ::* ;

pub fn sol_transfer (ctx : Context < SolTransfer

> , amount : u64) -> Result <()> { let from_pubkey = ctx . accounts . pda_account . to_account_info (); let to_pubkey = ctx . accounts . recipient . to_account_info (); let program_id = ctx . accounts . system_program . to_account_info ();

let seed = to_pubkey . key (); let bump_seed = ctx . bumps . pda_account; let signer_seeds : & [& [& [u8]]] = & [& [b"pda" , seed . as_ref (), & [bump_seed]]];

let cpi_context = CpiContext :: new (program_id, Transfer { from : from_pubkey, to : to_pubkey, },) . with_signer (signer_seeds);

transfer (cpi_context, amount) ? ; Ok (()) } }

[derive(

Accounts)] pub struct SolTransfer <' info

> {

[account(

mut , seeds = [b"pda" , recipient . key() . as_ref()], bump,)] pda_account : SystemAccount <' info

> ,

[account(

mut)] recipient : SystemAccount <' info

> , system_program : Program <' info , System , } Thecpi.test.ts file shows how to invoke the Anchor program'ssol_transfer instruction and logs a link to the transaction details on SolanaFM.

It shows how to derive the PDA using the seeds specified in the program:

const [PDA] = PublicKey. findProgramAddressSync ([Buffer. from (" pda "), wallet.publicKey . toBuffer ()], program.programId,); The first step in this example is to fund the PDA account with a basic SOL transfer from the Playground wallet.

cpi.test.ts it ("Fund PDA with SOL" , async () => { const transferInstruction = SystemProgram. transfer ({ fromPubkey: wallet.publicKey, toPubkey: PDA , lamports: transferAmount, });

const transaction = new Transaction (). add (transferInstruction);

const transactionSignature = await sendAndConfirmTransaction (connection, transaction, [wallet.payer], // signer);

console. log (\n Transaction Signature: + https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); }); Once the PDA is funded with SOL, invoke thesol_transfer instruction. This instruction transfers SOL from the PDA account back to thewallet account via a CPI to the System Program, which is "signed" for by the program.

it ("SOL Transfer with PDA signer" , async () => { const transactionSignature = await program.methods . solTransfer (new BN (transferAmount)) . accounts ({ pdaAccount: PDA , recipient: wallet.publicKey, }) . rpc ();

console. log (\n Transaction Signature: https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); }); You can build, deploy, and run the test to view the transaction details on the[SolanaFM explorer](https://solana.fm) .

The transaction details will show that the custom program was first invoked (instruction 1), which then invokes the System Program (instruction 1.1), resulting in a successful SOL transfer.

Transaction Details

Example 2 Explanation#

In the example code, theSolTransfer struct specifies the accounts required by the transfer instruction.

The sender is a PDA that the program must sign for. Theseeds to derive the address for thepda_account include the hardcoded string "pda" and the address of therecipient account. This means the address for thepda_account is unique for eachrecipient .

[derive(

Accounts)] pub struct SolTransfer <' info

{

[account(

mut , seeds = [b"pda" , recipient . key() . as_ref()], bump,)] pda_account : SystemAccount <' info

,

[account(

mut)] recipient : SystemAccount <' info

, system_program : Program <' info , System , } The Javascript equivalent to derive the PDA is included in the test file.

const [PDA] = PublicKey. findProgramAddressSync ([Buffer. from (" pda "), wallet.publicKey . toBuffer ()], program.programId,); The following tabs present two approaches to implementing Cross Program Invocations (CPIs), each at a different level of abstraction. Both examples are functionally equivalent. The main purpose is to illustrate the implementation details of the CPI.

1 2 Thesol_transfer instruction included in the example code shows a typical approach for constructing CPIs using the Anchor framework.

This approach involves creating a[CpiContext](https://docs.rs/anchor-lang) , which includes theprogram_id and accounts required for the instruction being called, followed by a helper function (transfer) to invoke a specific instruction.

pub fn sol_transfer (ctx : Context < SolTransfer

, amount : u64) -> Result <()> { let from_pubkey = ctx . accounts . pda_account . to_account_info (); let to_pubkey = ctx . accounts . recipient . to_account_info (); let program_id = ctx . accounts . system_program . to_account_info ();

let seed = to_pubkey . key (); let bump_seed = ctx . bumps . pda_account; let signer_seeds : & [& [& [u8]]] = & [& [b"pda" , seed . as_ref (), & [bump_seed]]];

let

cpi_context

= CpiContext :: new (program_id, Transfer { from : from_pubkey, to : to_pubkey, },) . with_signer (signer_seeds);

transfer (cpi_context , amount) ? ; Ok (()) } When signing with PDAs, the seeds and bump seed are included in thecpi_context assigner_seeds usingwith_signer() . The bump seed for a PDA can be accessed usingctx.bumps followed by the name of the PDA account.

let seed = to_pubkey . key (); let

bump_seed

= ctx . bumps . pda_account; let

signer_seeds : & [& [& [u8]]] = & [& [b"pda" , seed . as_ref (), & [bump_seed]]];

let cpi_context = CpiContext :: new (program_id, Transfer { from : from_pubkey, to : to_pubkey, },) . with_signer (signer_seeds); Thecpi_context andamount are then passed into thetransfer function to execute the CPI.

transfer (cpi_context, amount) ? ; When the CPI is processed, the Solana runtime will validate that the provided seeds and caller program ID derive a valid PDA. The PDA is then added as a signer on the invocation. This mechanism allows for programs to sign for PDAs that are derived from their program ID. Here is a reference program onSolana Playground which includes both examples.

Previous «PDAs with Anchor Next Rust Programs»