Program Derived Address

In this section, we'll walk through how to build a basic CRUD (Create, Read, Update, Delete) program. The program will store a user's message using a Program Derived Address (PDA) as the account's address.

The purpose of this section is to guide you through the steps for building and testing a Solana program using the Anchor framework and demonstrating how to use PDAs within a program. For more details, refer to thePrograms Derived Address page.

For reference, here is thefinal code after completing both the PDA and CPI sections.

Starter Code#

Begin by opening thisSolana Playground link with the starter code. Then click the "Import" button, which will add the program to your list of projects on Solana Playground.

Import

In thelib.rs file, you'll find a program scaffolded with thecreate ,update , anddelete instructions we'll implement in the following steps.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("8KPzbM2Cwn4Yjak7QYAEH9wyoQh86NcBicaLuzPaejdw");

[program]

pub mod pda { use super ::* ;

pub fn create (_ctx : Context < Create

) -> Result <()> { Ok (()) }

pub fn update (_ctx : Context < Update

) -> Result <()> { Ok (()) }

pub fn delete (_ctx : Context < Delete

) -> Result <()> { Ok (()) } }

[derive(

Accounts)] pub struct Create {}

[derive(

Accounts)] pub struct Update {}

[derive(

Accounts)] pub struct Delete {}

[account]

pub struct MessageAccount {} Before we begin, runbuild in the Playground terminal to check the starter program builds successfully.

Terminal build

Output

Define Message Account Type#

First, let's define the structure for the message account that our program will create. This is the data that we'll store in the account created by the program.

Inlib.rs , update theMessageAccount struct with the following:

lib.rs

[account]

pub struct MessageAccount { pub user : Pubkey , pub message : String , pub bump : u8 , }

Diff

Explanation

Build the program again by runningbuild in the terminal.

Terminal build We've defined what our message account will look like. Next, we'll implement the program instructions.

Implement Create Instruction#

Now, let's implement thecreate instruction to create and initialize theMessageAccount .

Start by defining the accounts required for the instruction by updating theCreate struct with the following:

lib.rs

[derive(

Accounts)]

[instruction(message

: String)] pub struct Create <' info

> {

[account(

mut)] pub user : Signer <' info

> ,

[account(

init, seeds = [b"message" , user . key() . as_ref()], bump, payer = user, space = 8 + 32 + 4 + message . len () + 1)] pub message_account : Account <' info , MessageAccount

> , pub system_program : Program <' info , System , }

Diff

Explanation

Next, implement the business logic for thecreate instruction by updating thecreate function with the following:

lib.rs pub fn create (ctx : Context < Create

> , message : String) -> Result <()> { msg! ("Create Message: {}" , message); let account_data = &mut ctx . accounts . message_account; account_data . user = ctx . accounts . user . key (); account_data . message = message; account_data . bump = ctx . bumps . message_account; Ok (()) }

Diff

Explanation

Rebuild the program.

Terminal build

Implement Update Instruction[#]

Next, implement theupdate instruction to update theMessageAccount with a new message.

Just as before, the first step is to specify the accounts required by theupdate instruction.

Update theUpdate struct with the following:

lib.rs

[derive(

Accounts)]

[instruction(message

: String)] pub struct Update <' info

> {

[account(

mut)] pub user : Signer <' info

> ,

[account(

mut , seeds = [b"message" , user . key() . as_ref()], bump = message_account . bump, realloc = 8 + 32 + 4 + message . len () + 1 , realloc :: payer = user, realloc :: zero = true ,)] pub message_account : Account <' info , MessageAccount

> , pub system_program : Program <' info , System , }

Diff

Explanation

Next, implement the logic for theupdate instruction.

lib.rs pub fn update (ctx : Context < Update

> , message : String) -> Result <()> { msg! ("Update Message: {}" , message); let account_data = &mut ctx . accounts . message_account; account_data . message = message; Ok (()) }

Diff

Explanation

Rebuild the program

Terminal build

Implement Delete Instruction[#]

Next, implement thedelete instruction to close theMessageAccount .

Update theDelete struct with the following:

lib.rs

[derive(

Accounts)] pub struct Delete <' info

{

[account(

mut)] pub user : Signer <' info

,

[account(

mut , seeds = [b"message" , user . key() . as_ref()], bump = message_account . bump, close = user,)] pub message_account : Account <' info , MessageAccount

, }

Diff

Explanation

Next, implement the logic for thedelete instruction.

lib.rs pub fn delete (_ctx : Context < Delete

) -> Result <()> { msg! ("Delete Message"); Ok (()) }

Diff

Explanation

Rebuild the program.

Terminal build

Deploy Program#

The basic CRUD program is now complete. Deploy the program by runningdeploy in the Playground terminal.

Terminal deploy

Output

Set Up Test File#

Included with the starter code is also a test file inanchor.test.ts .

anchor.test.ts import { PublicKey } from "@solana/web3.js" ;

describe ("pda" , () => { it ("Create Message Account" , async () => {});

it ("Update Message Account" , async () => {});

it ("Delete Message Account" , async () => {}); }); Add the code below insidedescribe , but before theit sections.

anchor.test.ts const program = pg.program; const wallet = pg.wallet;

const [messagePda , messageBump] = PublicKey. findProgramAddressSync ([Buffer. from ("message"), wallet.publicKey. toBuffer ()], program.programId,);

Diff

Explanation

Run the test file by runningtest in the Playground terminal to check the file runs as expected. We will implement the tests in the following steps.

Terminal test

Output

Invoke Create Instruction#

Update the first test with the following:

anchor.test.ts it ("Create Message Account" , async () => { const message = "Hello, World!" ; const transactionSignature = await program.methods . create (message) . accounts ({ messageAccount: messagePda, }) . rpc ({ commitment: "confirmed" });

const messageAccount = await program.account.messageAccount. fetch (messagePda, "confirmed" ,);

console. log (JSON . stringify (messageAccount, null , 2)); console. log ("Transaction Signature:" , https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); });

Diff

Explanation

Invoke Update Instruction#

Update the second test with the following:

anchor.test.ts it ("Update Message Account" , async () => { const message = "Hello, Solana!" ; const transactionSignature = await program.methods . update (message) . accounts ({ messageAccount: messagePda, }) . rpc ({ commitment: "confirmed" });

const messageAccount = await program.account.messageAccount. fetch (messagePda, "confirmed" ,);

console. log (JSON . stringify (messageAccount, null , 2)); console. log ("Transaction Signature:" , https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); });

Diff

Explanation

Invoke Delete Instruction#

Update the third test with the following:

anchor.test.ts it ("Delete Message Account" , async () => { const transactionSignature = await program.methods . delete () . accounts ({ messageAccount: messagePda, }) . rpc ({ commitment: "confirmed" });

const messageAccount = await program.account.messageAccount. fetchNullable (messagePda, "confirmed" ,);

console. log ("Expect Null:" , JSON . stringify (messageAccount, null , 2)); console. log ("Transaction Signature:" , https://solana.fm/tx{ transactionSignature }?cluster=devnet-solana ,); });

Diff

Explanation

Run Test#

Once the tests are set up, run the test file by runningtest in the Playground terminal.

Terminal test

Output

Previous «Deploying Programs Next Cross Program Invocation»