Rust Registration Q4 2024

After you have completed the Typescript Version of the Enrollment App, please do this one. This is only requited for all returning cadets and anyone new who optionally wants to do it.

Lesson One: Enrollment dApp In this lesson, we are going to:

Learn how to use solana-sdk to create a new keypair
Use our Public Key to airdrop ourselves some Solana devnet tokens Make Solana transfers on devnet

Empty your devnet wallet into your Turbin3 wallet
Use our Turbin3 Private Key to enroll in the Turbin3 enrollment dApp Prerequisites:

Have Rust and Cargo installed
Have a fresh folder created to follow this tutorial and all future tutorials Let's get into it!

1. Create a new Keypair

To get started, we're going to create a keygen script and an airdrop script for our account.

1.1 Setting up

Start by opening up your Terminal. We're going to use Cargo to initialise a new Rust library.

```
cargo init --lib
```

Now that we have our new project initialised, we're going to go ahead and add the solana-sdk to Cargo.toml.

```
[dependencies]
 solana-sdk = "1.15.2"
```

Finally, we're going to create some functions in our src/lib.rs file to let us run the three scripts we're going to build today, and annotate them with #[test] so we can easily call them:

```
#[cfg(test)] mod tests {

use solana_sdk;

#[test]
 fn keygen() {} #[test]
 fn airdop() {} #[test]
 fn transfer_sol() {}

}
```

Alright, we're ready to start getting into the code!

1.2 Generating a Keypair

We're going to create the keygen function to generate ourselves a new keypair. We'll start by importing Keypair, Signer and Pubkey from solana_sdk

use solana_sdk::{signature::{Keypair, Signer}, pubkey::Pubkey}; Now we're going to create a new Keypair, like so:

```
// Create a new keypair
 let kp = Keypair::new();
```
Now we're going to print out our new keypair to the console!

```
#[test]
 fn keygen() {


// Create a new keypair
 let kp = Keypair::new();
 println!("You've generated a new Solana wallet: {}", kp.pubkey().to_string()); println!("");
 println!("To save your wallet, copy and paste the following into a JSON file:");
println!("{:?}", kp.to_bytes());


}
```
Now we can click "Run test" on our keygen function, or execute cargo test keygen in our Terminal to generate a new keypair!

```
You've generated a new Solana wallet:
2sNvwMf15WPp94kywgvfn3KBNPNZhr5mWrDHmgjkjMhN


To save your wallet, copy and paste your private key into a JSON file:
[34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,62,153,150,99,255,20
2
,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,144,44
,28,56,231,114,50,131,185,168,138,61,35,98,78,53]
```

If we want to save this wallet locally, we're going to have to save it to a JSON file. To do this, we'll execute the following command:

```
touch dev-wallet.json
```
This creates the file dev-wallet.json in our root directory. Now we just need to paste the private key from above into this file, like so:

```
[34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,62,153,150,99,255,20
2
,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,144,44
,28,56,231,114,50,131,185,168,138,61,35,98,78,53]
```

Congrats, you've created a new Keypair and saved it your wallet. Let's go claim some tokens!

1.3 Import/Export to Phantom

Solana wallet files and wallets like Phantom use different encoding. While Solana wallet files use a byte array, Phantom uses a base58 encoded string representation of private keys. To go between these formats, we can use the bs58 crate. Go ahead and add bs58 to your Cargo.toml, then add imports for use::bs58 and use std::io::{self, BufRead}:

```
use bs58;
 use std::io::{self, BufRead};
```

Now add in the following two convenience functions to your tests and you should have a simple CLI tool to convert between wallet formats:

```
#[test]
 fn base58_to_wallet() {


println!("Input your private key as base58:");
 let stdin = io::stdin();
 let base58 = stdin.lock().lines().next().unwrap().unwrap(); println!("Your wallet file is:");
 let wallet = bs58::decode(base58).into_vec().unwrap(); println!("{:?}", wallet);


}
```

```
#[test]
 fn wallet_to_base58() {


println!("Input your private key as a wallet file byte array:"); let stdin = io::stdin();
 let wallet =


stdin.lock().lines().next().unwrap().unwrap().trim_start_matches('[').trim_end_matches(']').
split(',') .map(|s| s.trim().parse::<u8>().unwrap()).collect::<Vec<u8>>();


println!("Your private key is:");
 let base58 = bs58::encode(wallet).into_string(); println!("{:?}", base58);


}
```

You can try going between these two private key types to test your code is working:

```
// Wallet file
 //
[34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,62,153,150,99,255,20
2
,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,144,44
,28,56,231,114,50,131,185,168,138,61,35,98,78,53]


// Base 58

//
gdtKSTXYULQNx87fdD3YgXkzVeyFeqwtxHm6WdEb5a9YJRnHse7GQr7t5pbepsyvUCk7V
vks UGhPt4SZ8JHVSkt
```

2. Claim Token Airdrop

Now that we have our wallet created, we're going to import it into another script. Start by adding solana_client to our Cargo.toml.

solana-client = "1.15.2"
We're going to need this to import RpcClient to let us establish a connection to the Solana devnet, and read_keypair_file which lets us import our wallet from a wallet file. Our imports will now be as follows:

use solana_client::rpc_client::RpcClient; use solana_sdk::{

signature::{Keypair, Signer, read_keypair_file} };

Now we're going to add in a const at the top of our file underneath our imports for the devnet API url:

const RPC_URL: &str = "https://api.devnet.solana.com";
Okay, we're ready to get started. Let's go to the airdrop function and start by reading in our keypair file:

// Import our keypair
let keypair = read_keypair_file("dev-wallet.json").expect("Couldn't find wallet file"); Then we'll establish a connection to Solana devnet using the const we defined above:

// Connected to Solana Devnet RPC Client
let client = RpcClient::new(RPC_URL); Finally, we're going to call the airdrop function:

// We're going to claim 2 devnet SOL tokens (2 billion lamports)
match client.request_airdrop(&keypair.pubkey(), 2_000_000_000u64) {

Ok(s) => {
println!("Success! Check out your TX here:");
println!("https://explorer.solana.com/tx/{}?cluster=devnet", s.to_string());

},

Err(e) => println!("Oops, something went wrong: {}", e.to_string()) };

Here is an example of the output of a successful airdrop: Success! Check out your TX here:

https://explorer.solana.com/tx/459QHLHJBtkHgV3BkzGKo4CDSWzNr8HboJhiQhpx2dj8xPVqx4 BtUPCDWYbbTm426mwqmdYBhEodUQZULcpvzd5z?cluster=devnet

3. Transfer tokens to your Turbin3 Address

Now we have some devnet SOL to play with, it's time to create our first native Solana token transfer. When you first signed up for the course, you gave Turbin3 a Solana address for certification. We're going to be sending some devnet SOL to this address so we can use it going forward.

First, we're going to add solana-program to our Cargo.toml.

solana-program = "1.15.2"
Then we're going to add a few more imports to let us create a transfer transaction:

use solana_client::rpc_client::RpcClient; use solana_program::{

pubkey::Pubkey,

system_instruction::transfer, };

use solana_sdk::{
signature::{Keypair, Signer, read_keypair_file}, transaction::Transaction

};
 use std::str::FromStr;
 Now let's open up the transfer_sol function, import our dev wallet as we did last time, and define our Turbin3 public key:

// Import our keypair
 let keypair = read_keypair_file("dev-wallet.json").expect("Couldn't find wallet file");

// Define our Turbin3 public key
 let to_pubkey = Pubkey::from_str("<your Turbin3 public key>").unwrap(); Now let's create a connection to devnet

// Create a Solana devnet connection
 let rpc_client = RpcClient::new(RPC_URL);
 In order to sign transactions, we're going to need to get a recent blockhash, as signatures are designed to expire as a security feature:

// Get recent blockhash
 let recent_blockhash = rpc_client .get_latest_blockhash()
 .expect("Failed to get recent blockhash");

Okay, we now have everything we need to create and sign our transaction! We're going to transfer 0.1 SOL from our dev wallet to our Turbin3 wallet address on the Solana devnet.

let transaction = Transaction::new_signed_with_payer(&[transfer(

&keypair.pubkey(), &to_pubkey, 1_000_000

)], Some(&keypair.pubkey()), &vec![&keypair], recent_blockhash

);
 Now that we've signed out transaction, we're going to submit our transaction and grab the TX signature

// Send the transaction
 let signature = rpc_client
 .send_and_confirm_transaction(&transaction)
 .expect("Failed to send transaction");
 If everything went as planned, we'll print a link to the TX out to the terminal

// Print our transaction out println!(

"Success! Check out your TX here: https://explorer.solana.com/tx/{}/?cluster=devnet", signature

);

4. Empty devnet wallet into Turbin3 wallet

Okay, now that we're done with our devnet wallet, let's also go ahead and send all of our remaining lamports to our Turbin3 dev wallet. It is typically good practice to clean up accounts where we can as it allows us to reclaim resources that aren't being used which have actual monetary value on mainnet.

To send all of the remaining lamports out of our dev wallet to our Turbin3 wallet, we're going to need to add in a few more lines of code to the above example so we can:

Get the exact balance of the account
 Calculate the fee of sending the transaction
 Calculate the exact number of lamports we can send whilst satisfying the fee rate Start by adding Message to our imports.

```
use solana_sdk::{

message::Message,
 signature::{Keypair, Signer, read_keypair_file}, transaction::Transaction



};
```

To empty the account, we're going to have to find out how much balance it has. We can do that like so:

```
// Get balance of dev wallet
 let balance = rpc_client
 .get_balance(&keypair.pubkey())
 .expect("Failed to get balance");
```

Now that we have the balance, we need to calculate the current fee rate for sending a transaction on devnet. To do that, we need to make a mock transaction and ask the RPC client how much it would cost to publish. We'll start by making the mock transaction as a Message.

```
// Create a test transaction to calculate fees
 let message = Message::new_with_blockhash(



&[transfer( &keypair.pubkey(), &to_pubkey, balance,



)], Some(&keypair.pubkey()), &recent_blockhash



);
```

Now we need to ask the RPC Client what the fee for this mock transaction would be:

```
// Calculate exact fee rate to transfer entire SOL amount out of account minus fees let fee
= rpc_client
```

```
 .get_fee_for_message(&message)
 .expect("Failed to get fee calculator");
```

Now that we have the balance and the fee, it's simply a matter of copying the transaction code from above and replacing the 1_000_000 lamports with the value of balance - fee:

```
// Deduct fee from lamports amount and create a TX with correct balance let transaction =
Transaction::new_signed_with_payer(

&[transfer( &keypair.pubkey(), &to_pubkey, balance - fee,

)], Some(&keypair.pubkey()), &vec![&keypair], recent_blockhash);
```

As you can see, we crated a mock version of the transaction to perform a fee calculation before removing and readding the transfer instruction, signing and sending it. You can see from the outputted transaction signature on the block explorer here that the entire value was sent to the exact lamport:

Check out your TX here:

https://explorer.solana.com/tx/4dy53oKUeh7QXr15wpKex6yXfz4xD2hMtJGdqgzvNnYyDNBZXtc gKZ7NBvCj7PCYU1ELfPZz3HEk6TzT4VQmNoS5?cluster=devnet

5. Submit your completion of the Turbin3 pre-requisites program

When you first signed up for the course, you gave Turbin3 a Solana address for certification and your Github account. Your challenge now is to use the devnet tokens you just airdropped and transferred to yourself to confirm your enrollment in the course on the Solana devnet.

In order to do this, we're going to have to quickly familiarize ourselves with two key concepts of Solana:

PDA (Program Derived Address) - A PDA is used to enable our program to "sign" transactions with a Public Key derived from some kind of deterministic seed. This is then combined with an additional "bump" which is a single additional byte that is generated to "bump" this Public Key off the elliptic curve. This means that there is no matching Private Key for this Public Key, as if there were a matching private key and someone happened to possess it, they would be able to sign on behalf of the program, creating security concerns.

IDL (Interface Definition Language) - Similar to the concept of ABI in other ecosystems, an IDL specifies a program's public interface. Though not mandatory, most programs on Solana do have an IDL, and it is the primary way we typically interact with programs on Solana. It defines a Solana program's account structures, instructions, and error codes. IDLs are .json files, so they can be used to generate client-side code, such as Typescript type definitions, for ease of use.

Let's dive into it!

5.1 Consuming an IDL in Typescript

For the purposes of this class, we have published a Turbin3 pre-requisite course program to the Solana Devnet with a public IDL that you can use to provide onchain proof that you've made it to the end of our pre-requisite coursework.

You can find our program on Devnet by this address: HC2oqz2p6DEWfrahenqdq2moUcga9c9biqRBcdK3XKU1

If we explore the devnet explorer, there is a tab called "Anchor Program IDL" which reveals the IDL of our program. If you click the clipboard icon at the top level of this

JSON object, you can copy the IDL directly from the browser. The result should look something like this:

```
{
 "version": "0.1.0", "name": "Turbin3_prereq", "instructions": [



{
 "name": "complete", ...



} ]



}
```

As you can see, this defines the schema of our program with a single instruction called complete that takes in 1 argument:

github - a byte representation of the utf8 string of your github account name As well as 3 accounts:

signer - your public key you use to sign up for the Turbin3 course
prereq - an account we create in our program with a custom PDA seed (more on this later) systemAccount - the Solana system program which is used to execute account instructions In order for us to consume this in rust, we're going to go and use the solana-idlgen macro to generate Rust code for it. Let's start off by adding these two imports to our Cargo.toml:

```
borsh = "0.10.3"
 solana-idlgen = { git = "https://github.com/deanmlittle/solana-idlgen.git" }
```

Next, we must create a folder in our src directory called programs so we can easily add additional program IDLs in the future, along with two new rust files called mod.rs and Turbin3_prereq.rs.

```
mkdir src/programs
 touch ./src/programs/mod.rs
```

touch ./sc/programs/Turbin3_prereq.rs
Now we need to publicly declare our Turbin3_prereq module in our programs module, mod.rs:

```
// Programs
 pub mod Turbin3_prereq;
```

Now we need to import and use the IDLGen macro to consume our IDL. Open up Turbin3_prereq.rs and input our IDL Json into the idlgen! macro:

```
use solana_idlgen::idlgen; idlgen!({

"version": "0.1.0", "name": "Turbin3_prereq", "instructions": [{
 "name": "complete", ...
 }

] });
```

We will also need to populate the optional metadata: { address } fields of the IDL:

```
idlgen!({
 "version": "0.1.0", "name": "Turbin3_prereq", ...
 "metadata": {

"address": "HC2oqz2p6DEWfrahenqdq2moUcga9c9biqRBcdK3XKU1" }

});
```

You should now have a working Turbin3PrereqProgram library generated at build time! Now let's import it. We start by adding our programs module to the top of lib.rs, outside of our test module, so we can consume it from our crate.

mod programs;
Then inside our tests module, we will add the imports for the generated structs and methods we need to use:

use crate::programs::Turbin3_prereq::{Turbin3PrereqProgram, CompleteArgs, UpdateArgs}; Now it's time to put it all together! As with last time, we need to create a Solana devnet connection:

// Create a Solana devnet connection
let rpc_client = RpcClient::new(RPC_URL);
Now let's define our accounts. For this exercise, we will be using our Turbin3 wallet. Unlike the dev-wallet.json we generated, we care about maintaining the security of this private key. To stop you from accidentally comitting your private key(s) to a git repo, consider adding a .gitignore file. Here's an example that will ignore all files that end in wallet.json:

***wallet.json**
Okay, now we can add our Turbin3 wallet:

// Let's define our accounts
let signer = read_keypair_file("Turbin3-wallet.json").expect("Couldn't find wallet file"); Next up, we'll be creating our first PDA.

5.2 Creating a PDA

Now we need to create a PDA for our prereq account. The seeds for this particular PDA are:

A u8 array of the text: "prereq"
 A u8 array of the Pubkey of the transaction signer
 These are then combined into an array of u8 array pointers and combined with the program ID to create a deterministic address for this account. The derive_program_address function is going to combine this with the program id and calculate a bump for it to find an address that is not on the elliptic curve and then return the derived address:

Remember to familiarize yourself with this concept as you'll be using it often!

```
let prereq = Turbin3PrereqProgram::derive_program_address(&[b"prereq",
signer.pubkey().to_bytes().as_ref()]);
```

5.3 Putting it all together

Now that we have everything we need, it's finally time to put it all together and make a transaction interacting with the devnet program to submit our github account and our Pubkey to signify our completion of the Turbin3 pre-requisite materials!

We have to populate our instruction data. In this case, we are setting your Github account username:

```
// Define our instruction data let args = CompleteArgs {

github: b"testaccount".to_vec() };
```

To publish our transaction, as we last time, we need a recent block hash:

```
// Get recent blockhash
 let blockhash = rpc_client .get_latest_blockhash()
 .expect("Failed to get recent blockhash");
```

Now we need to populate our complete function:

```
// Now we can invoke the "complete" function let transaction =
Turbin3PrereqProgram::complete(

&[&signer.pubkey(), &prereq, &system_program::id()], &args,
 Some(&signer.pubkey()),
 &[&signer],

blockhash );
```

Finally, we publish our transaction! // Send the transaction

```
let signature = rpc_client .send_and_confirm_transaction(&transaction) .expect("Failed
to send transaction");

// Print our transaction out
 println!("Success! Check out your TX here:
https://explorer.solana.com/tx/{}/?cluster=devnet", signature);
```

Congratulations, now we get to work.