**Prerequisites: Enrollment dApp - Rust**
**READ CAREFULLY ALL STEPS**
**(NOTE-If you are a windows user, you will want to work in WSL2 for Solana)**
**You have 24 hours to complete this.**

*These prerequisites are meant to assess your ability to follow processes, execute tasks, debug simple errors (intentionally placed), and ship code. They are not a test of your typescript or coding skills. It is integral that you UNDERSTAND what is happening every step of the way. This is a foundational process that we jump right into building upon in week one of the cohort. As always, when in doubt- after trying for a minute- ask in the Solana Chat channel of the Discord. If this is beyond your capacity at the moment, we will help you find the right program to better prepare you for a future cohort.*

**Prerequisites:**
Have Rust installed (rustup)
Have Cargo installed
Have a fresh folder created to follow this tutorial and all future tutorials

Now we start - you will:

- Learn how to use the solana-sdk to create a new keypair
- Use your Public Key to airdrop some Solana devnet tokens
- Make Solana transfers on devnet
- Empty your devnet wallet into your Turbin3 wallet
- Use your Turbin3 Private Key to interact with the Turbin3 enrollment dApp
- Reuse your Solana on-chain account created during TypeScript prerequisites for your Turbin3 Application
- Mint yourself a NFT that proves your successful completion of the Turbin3 Pre Reqs (Rust)

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

Step 1: Set up the Rust project, start by opening up your Terminal. We're going to use Cargo to create a new Rust project.

*command: cargo init --lib*

Step 2: Add dependencies, Now that we have our new project initialized, we're going to go ahead and add the solana-sdk to the Cargo.toml.

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

Step 3:Create scaffolding in src/lib.rs

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 airdrop() {}

#[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.

Step 1: Import required dependencies

```
 use solana_sdk::{signature::{Keypair, Signer}, pubkey::Pubkey};
```

Now we're going to create a new Keypair.

Step 2: Create a new keypair

```
 // Create a new keypair
let kp = Keypair::new();
```

Step 3: Print the wallet and its private key

```
 #[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
,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,14
4,44 ,28,56,231,114,50,131,185,168,138,61,35,98,78,53]

Step 4: Create the wallet file
To store your wallet locally, you'll want to save it to a JSON file.

Run the following command in your terminal:

*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
,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,14
4,44 ,28,56,231,114,50,131,185,168,138,61,35,98,78,53]*

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

### 1.3. Import/Export to Phantom

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.

Solana wallet files and wallets like Phantom use different encodings.
While Solana wallet files use a raw byte array (JSON array of numbers), Phantom uses a Base58-encoded string to represent private keys.

Step 1: Add bs58 crate to Cargo.toml
To convert between these formats, you can use the bs58 crate.

*bs58 = "0.4"*

Step 2: Import the necessary dependencies:

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

Step 3: Function to convert Base58 string (Phantom) to wallet file format (JSON byte array)

```
#[test]
fn base58_to_wallet() {
    println!("Input your private key as a base58 string:");
    let stdin = io::stdin();
    let base58 = stdin.lock().lines().next().unwrap().unwrap();
    println!("Your wallet file format is:");
    let wallet = bs58::decode(base58).into_vec().unwrap();
    println!("{:?}", wallet);
}
```

Step 4: Function to convert wallet file format (JSON byte array) to Base58 string (Phantom)

```
#[test]
fn wallet_to_base58() {
    println!("Input your private key as a JSON byte array (e.g. [12,34,...]):");
    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 Base58-encoded private key is:");
```

```
    let base58 = bs58::encode(wallet).into_string();

    println!("{:?}", base58);

}
```

Step 5: Test converting between formats. You can test by converting between these two formats:

// Wallet file format

*[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*
*,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,14 4,44 ,28,56,231,114,50,131,185,168,138,61,35,98,78,53]*

// Base58 format

*"gdtKSTXYULQNx87fdD3YgXkzVeyFeqwtxHm6WdEb5a9YJRnHse7GQr7t5pbepsyvUCk7V VksUGhPt4SZ8JHVSkt"*

**2. Claim Token Airdrop**

Step 1: Add the solana-client crate to 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.

Step 2: Import dependencies

```
use solana_client::rpc_client::RpcClient;
use solana_sdk::{
    signature::{Keypair, Signer, read_keypair_file},
};
```

Step 3: Define the devnet RPC endpoint constant

const RPC_URL: &str = "https://turbine-solanad-4cde.devnet.rpcpool.com/9a9da9cf-6db1-47dc-839a-55aca5c9c80a";

Step 4: Create function to claim airdrop

```
fn claim_airdrop() {
    // Import our keypair
```

```
    let keypair = read_keypair_file("dev_wallet.json").expect("Couldn't find wallet file");

    // we'll establish a connection to Solana devnet using the const we defined above
    let client = RpcClient::new(RPC_URL);

    // We're going to claim 2 devnet SOL tokens (2 billion lamports)
    match client.request_airdrop(&keypair.pubkey(), 2_000_000_000u64) {
        Ok(sig) => {
            println!("Success! Check your TX here:");
            println!("https://explorer.solana.com/tx/{}?cluster=devnet", sig);
        }
        Err(err) => {
            println!("Airdrop failed: {}", err);
        }
    }
}
```

### 3. Transfer tokens to your Turbin3 wallet

Now that we have some devnet SOL to play with, it's time to create our first native Solana token transfer in Rust. We're going to send some devnet SOL to the Turbin3 Solana Address you have registered with so we can use it going forward.

Step 1: Add the solana-program crate to Cargo.toml

In your Cargo.toml file add the following line:

```
solana-program = "1.15.2"
```

Step 2: Import necessary dependencies

Add these imports to your Rust file:

```
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;
```

Step 3: Load your dev wallet and verify the keypair

```
// Load your devnet keypair from file
let keypair = read_keypair_file("dev-wallet.json").expect("Couldn't find wallet file");

// Generate a signature from the keypair
let pubkey = keypair.pubkey();
```

```
let message_bytes = b"I verify my Solana Keypair!";
let sig = keypair.sign_message(message_bytes);
let sig_hashed = hash(sig.as_ref());

// Verify the signature using the public key
match sig.verify(&pubkey.to_bytes(), &sig_hashed.to_bytes()) {
    true => println!("Signature verified"),
    false => println!("Verification failed"),
}
```

Step 4: Define the destination (Turbin3) address

```
let to_pubkey = Pubkey::from_str("<your Turbin3 public key>").unwrap();
```

Step 5: Connect to devnet

```
let rpc_client = RpcClient::new(RPC_URL);
```

Step 6: Fetch 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.

Step 7: Create and sign the transaction

```
let transaction = Transaction::new_signed_with_payer(
    &[transfer(&keypair.pubkey(), &to_pubkey, 1_000_000)],
    Some(&keypair.pubkey()),
    &vec![&keypair],
    recent_blockhash,
);
```

Step 8: Send the transaction and print tx

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

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

**4. Empty your devnet wallet into your Turbin3 Wallet**

Now that we've tested the transfer, let's clean up the dev wallet by sending all remaining SOL (lamports) to the Turbin3 wallet.

This is a good practice for mainnet: it frees up resources (accounts) and avoids wasted SOL.

Step 1: Add Message to imports

```
use solana_sdk::{message::Message, signature::{Keypair, Signer, read_keypair_file},
transaction::Transaction};
```

Step 2: Get current balance

```
let balance = rpc_client
    .get_balance(&keypair.pubkey())
    .expect("Failed to get balance");
```

Step 3: Build a mock transaction to calculate fee

```
let message = Message::new_with_blockhash(
    &[transfer(&keypair.pubkey(), &to_pubkey, balance)],
    Some(&keypair.pubkey()),
    &recent_blockhash,
);
```

Step 4: Estimate transaction fee

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

Step 5: Create final transaction with balance minus fee

```
let transaction = Transaction::new_signed_with_payer(
    &[transfer(&keypair.pubkey(), &to_pubkey, balance - fee)],
    Some(&keypair.pubkey()),
    &vec![&keypair],
    recent_blockhash,
);
```

This ensures we leave zero lamports behind.

Step 6: Send transaction and verify

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

println!(
    "Success! Entire balance transferred: https://explorer.solana.com/tx/{}/?cluster=devnet",
    signature
```

*);*

## 5. Submit your completion of the Turbin3 prerequisites program

You have now reached one of the most important steps in the Turbin3 admission process! This part requires you to interact on-chain with the Turbin3 enrollment dApp by submitting proof that you have completed the prerequisite coursework.

Important:
You do not need to rerun the initialization step since it was already completed in the TypeScript prerequisites, which created the necessary on-chain PDA account. Now, your task is to submit the completion proof by calling the submit_rs function on the Rust side, which uses the same accounts as the submit_ts instruction in TypeScript.

---

**Key Concepts Recap (to help you understand the process)**

- **PDA (Program Derived Address):**
  This is a special kind of Solana address derived deterministically by the program from seeds and a bump. It allows the program to "sign" transactions without a private key, enhancing security.

- **IDL (Interface Definition Language):**
  Similar to ABI in Ethereum, an IDL defines the public interface of a Solana program (accounts, instructions, errors). This program's IDL helps you understand how to call its functions properly.

### 5.1. Turbin3 Program on Solana Devnet and Public IDL

There is a Turbin3 program published on the Solana devnet with a public IDL that you will need to interact with to provide on-chain proof that you've made into the end of the prerequisite coursework.

You can find our program on devnet by this address:
TRBZyQHB3m68FGeVsqTK39Wm4xejadjVhP5MAZaKWDM

And find the IDL of the program, that defines the schema of our program, here:
https://explorer.solana.com/address/TRBZyQHB3m68FGeVsqTK39Wm4xejadjVhP5MAZaKWDM/anchor-program?cluster=devnet

You will need the following instruction to complete your submission:

- submit_rs

This instruction receives the following accounts:
user – Your public key (used for the Turbin3 application).
account – The account created by the program (from the TypeScript prerequisite).

mint – The address of the new asset (created by the program).
collection – The collection to which the asset belongs.
authority – The authority signing the NFT creation (PDA).
mpl_core_program – The Metaplex Core program
system_program – The Solana system program

### 5.2. Send Raw Instruction to the Turbin3 Program in Rust

To interact with the Turbin3 program in Rust, you will manually construct the instruction using the known instruction discriminator and account metadata.

Step 1: Create a Solana RPC client

```
let rpc_client = RpcClient::new(RPC_URL);
```

Step 2: Load your signer keypair

Make sure to use the correct path for your wallet file. This keypair will sign and pay for the transaction.

```
let signer = read_keypair_file("dev-wallet.json")
    .expect("Couldn't find wallet file");
```

Step 3: Define program and account public keys

Specify all the public keys of the program and accounts your instruction will interact with.

```
let mint = Keypair::new();

let turbin3_prereq_program =
Pubkey::from_str("TRBZyQHB3m68FGeVsqTK39Wm4xejadjVhP5MAZaKWDM").unwrap();

let collection =
Pubkey::from_str("5ebsp5RChCGK7ssRZMVMufgVZhd2kFbNaotcZ5UvytN2").unwrap();

let mpl_core_program =
Pubkey::from_str("CoREENxT6tW1HoK8ypY1SxRMZTcVPm7R94rH4PZNhX7d").unwrap();

let system_program = system_program::id();
```

Step 4: Get the PDA (Program Derived Address)

We already created this PDA earlier in our TypeScript enrollment. Now we're replicating the same logic in Rust.

- The byte array of the string "prereqs"

- The byte array of the signer's public key

These are combined and used along with the program ID to create a deterministic address for the account:

*let signer_pubkey = signer.pubkey();*

*let seeds = &[b"prereqs", signer_pubkey.as_ref()];*

*let (prereq_pda, _bump) = Pubkey::find_program_address(seeds, &turbin3_prereq_program);*

Step 5: Prepare the instruction data (discriminator)

The discriminator uniquely identifies the instruction your program expects.
From the IDL, the submit_rs instruction discriminator is:

*let data = vec![77, 124, 82, 163, 21, 133, 181, 206];*

Step 6:  Define the accounts metadata

*let accounts = vec![*

*AccountMeta::new(signer.pubkey(), true),  // user signer*

*AccountMeta::new(prereq_pda, false),  // PDA account*

*AccountMeta::new(mint.pubkey(), true),  // mint keypair*

*AccountMeta::new(collection, false),  // collection*

*AccountMeta::new_readonly(authority, false),  // authority (PDA)*

*AccountMeta::new_readonly(mpl_core_program, false), // mpl core program*

*AccountMeta::new_readonly(system_program, false),  // system program*

*];*

Use new for accounts that the instruction writes to and new_readonly for accounts that are read-only. The true flag indicates the account must sign the transaction.

Step 7: Get the recent blockhash

We need a recent blockhash to build the transaction:

let blockhash = rpc_client

.get_latest_blockhash()

```
    .expect("Failed to get recent blockhash");
```

Step 8:

Build the instruction

Construct the instruction by specifying the program ID, accounts, and instruction data.

```
let instruction = Instruction {
    program_id: turbin3_prereq_program,
    accounts,
    data,
};
```

Step 9: Create and sign the transaction

Create a transaction containing the instruction and sign it with the necessary keypairs.

```
let transaction = Transaction::new_signed_with_payer(
    &[instruction],
    Some(&signer.pubkey()),
    &[&signer, &mint],
    blockhash,
);
```

Step 10: Send and confirm the transaction

```
let signature = rpc_client
    .send_and_confirm_transaction(&transaction)
    .expect("Failed to send transaction");
println!(
    "Success! Check out your TX here:\nhttps://explorer.solana.com/tx/{}/?cluster=devnet",
    signature
);
```

Congratulations! You are now ready to submit your proof of completion on-chain with the Rust submit_rs instruction, advancing further into your Turbin3 Builders Cohort admission.