

Prerequisites: Enrollment dApp

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.

Now we start- you will:

- Learn how to use @solana/web3.js 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 NodeJS installed

Have yarn 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 yarn to create a new Typescript project.

```
mkdir airdrop && cd airdrop
yarn init -y
```

Now that we have our new project initialized, we're going to go ahead and add typescript, bs58 and @solana/web3.js, along with generating a tsconfig.js configuration file.

```
yarn add @types/node typescript @solana/web3.js bs58
yarn add -D ts-node
touch keygen.ts
touch airdrop.ts
touch transfer.ts
touch enroll.ts
```

```
yarn tsc --init --rootDir ./ --outDir ./dist --esModuleInterop --lib ES2019 --module commonjs --resolveJsonModule true --noImplicitAny true
```

Finally, we're going to create some scripts in our package.json file to let us run the three scripts we're going to build today:

```
{
  "name": "airdrop",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "keygen": "ts-node ./keygen.ts",
    "airdrop": "ts-node ./airdrop.ts",
    "transfer": "ts-node ./transfer.ts",
    "enroll": "ts-node ./enroll.ts"
  },
  "dependencies": {
    "@solana/web3.js": "^1.75.0",
    "@types/node": "^18.15.11",
    "typescript": "^5.0.4"
  },
  "devDependencies": {
    "ts-node": "^10.9.1"
  }
}
```

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

1.2 Generating a Keypair

Open up ./keygen.ts. We're going to generate ourselves a new keypair.

We'll start by importing Keypair from @solana/web3.js

```
import { Keypair } from "@solana/web3.js";
Now we're going to create a new Keypair, like so:

//Generate a new keypair
let kp = Keypair.generate()
```

```
console.log(`You've generated a new Solana wallet:
${kp.publicKey.toBase58()}`)
```

To save your wallet, copy and paste the output of the following into a JSON file:

```
console.log(`[${kp.secretKey}]`)
```

Now we can run the following script in our terminal to generate a new keypair!

```
yarn keygen
```

This will generate a new Keypair, outputting its Address and Private Key like so:

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

To save your wallet, copy and paste your private key into a JSON file:

```
[34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,6
2
,153,150,99,255,202,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,
132,234,160,203,109,195,116,251,144,44,28,56,231,114,50,131,185,168,13
8,61,35,98,78,53]
```

If we want to save this wallet locally. To do this, we're going to run the following

```
command: touch dev-wallet.json
```

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

```
[34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,6
2
,153,150,99,255,202,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,
132,234,160,203,109,195,116,251,144,44,28,56,231,114,50,131,185,168,13
8,61,35,98,78,53]
```

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

1.3 Import/Export to Phantom

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. If you would like to go between these formats, try import the bs58 package. We'll also use the prompt-sync package to take in the private key:

```
yarn add bs58 prompt-sync
```

```
import bs58 from 'bs58'
import * as prompt from 'prompt-sync'
```

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!("Enter your name:");
    let stdin = io::stdin();
    let base58 = stdin.lock().lines().next().unwrap().unwrap(); //

    gdtKSTXYULQNx87fdD3YgXkzVeyFeqwtxHm6WdEb5a9YJRnHse7GQr7t5pbepsyvUCk7Vv
    ksUGhPt4SZ8JHVSkt

    let wallet = bs58::decode(base58).into_vec().unwrap();
    println!("{:?}", wallet);
}

#[test]
fn wallet_to_base58() {
    let wallet: Vec<u8> =
    vec![34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,6
```

```
3,62,153,150,99,255,202,89,105,77,41,89,253,130,27,195,134,14,66,75,24
2,7,132,234,160,203,109,195,116,251,144,44,28,56,231,114,50,131,185,16
8,138,61,35,98,78,53];
```

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

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

2. Claim Token Airdrop

Now that we have our wallet created, we're going to import it into another script

This time, we're going to import Keypair, but we're also going to import Connection to let us establish a connection to the Solana devnet, and LAMPORTS_PER_SOL which lets us conveniently send ourselves amounts denominated in SOL rather than the individual lamports units.

```
import { Connection, Keypair, LAMPORTS_PER_SOL } from
"@solana/web3.js"
```

We're also going to import our wallet and recreate the Keypair object using its private key:

```
import wallet from "./dev-wallet.json"
```

```
// We're going to import our keypair from the wallet file
```

```
const keypair = Keypair.fromSecretKey(new Uint8Array(wallet));
```

Now we're going to establish a connection to the Solana devnet:

```
//Create a Solana devnet connection to devnet SOL tokens
```

```
const connection = new Connection("https://api.devnet.solana.com");
```

Finally, we're going to claim 2 devnet SOL tokens:

```
(async () => {
  try {
    // We're going to claim 2 devnet SOL tokens
    const txhash = await
```

```

connection.requestAirdrop(keypair.publicKey, 2 * LAMPORTS_PER_SOL);
    console.log(`Success! Check out your TX here:
    https://explorer.solana.com/tx/${txhash}?cluster=devnet`);
  } catch(e) {
    console.error(`Oops, something went wrong: ${e}`)
  }
})();

```

Here is an example of the output of a successful airdrop:

Success! Check out your TX here:

<https://explorer.solana.com/tx/459QHLHJBtkHgV3BkzGKo4CDSWzNr8HboJhiQhpX2dj8xPVqx4BtUPCDWYbbTm426mwqmdYBhEodUQZULcpvzd5z?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. (NOTE: If you did not enter the address in the application, or you are using a different address, you need to reach out to Jeff, ASAP).

We're going to open up transfer.ts and import the following items from @solana/web3.js:

```

import { Transaction, SystemProgram, Connection, Keypair,
LAMPORTS_PER_SOL, sendAndConfirmTransaction, PublicKey } from
"@solana/web3.js"

```

We will also import our dev wallet as we did last time:

```

// Import our dev wallet keypair from the wallet file

const from = Keypair.fromSecretKey(new Uint8Array(wallet));

// Define our Turbin3 public key
const to = new
PublicKey("GLtaTaYiTQrgz411iPJJD79rsoee59HhEy18rtRdrhEUJ");

```

And create a devnet connection:

```

//Create a Solana devnet connection

```

```
const connection = new Connection("https://api.devnet.solana.com");
```

Now we're going to create a transaction using @solana/web3.js to transfer 0.1 SOL from our dev wallet to our Turbin3 wallet address on the Solana devnet. Here's how we do that:

```
(async () => {
  try {
    const transaction = new Transaction().add(
      SystemProgram.transfer({
        fromPubkey: from.publicKey,
        toPubkey: to,
        lamports: LAMPORTS_PER_SOL/100,
      })
    );
    transaction.recentBlockhash = (await
connection.getLatestBlockhash('confirmed')).blockhash;
    transaction.feePayer = from.publicKey;

    // Sign transaction, broadcast, and confirm
    const signature = await sendAndConfirmTransaction(
      connection,
      transaction,
      [from]
    );
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signature}?cluster=devnet`);
  } catch(e) {
    console.error(`Oops, something went wrong: ${e}`)
  }
})();
```

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 examples 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

```
(async () => {
  try {
    // Get balance of dev wallet
    const balance = await connection.getBalance(from.publicKey)

    // Create a test transaction to calculate fees
    const transaction = new Transaction().add(
      SystemProgram.transfer({
        fromPubkey: from.publicKey,
        toPubkey: to,
        lamports: balance,
      })
    );
    transaction.recentBlockhash = (await
connection.getLatestBlockhash('confirmed')).blockhash;
    transaction.feePayer = from.publicKey;

    // Calculate exact fee rate to transfer entire SOL amount out
of account minus fees
    const fee = (await
connection.getFeeForMessage(transaction.compileMessage(),
'confirmed')).value || 0;

    // Remove our transfer instruction to replace it
    transaction.instructions.pop();

    // Now add the instruction back with correct amount of
lamports
    transaction.add(
      SystemProgram.transfer({
        fromPubkey: from.publicKey,
        toPubkey: to,
        lamports: balance - fee,
      })
    );
  }
});
```



```

// Sign transaction, broadcast, and confirm
const signature = await sendAndConfirmTransaction(
  connection,
  transaction,
  [from]
);
console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signature}?cluster=devnet`)
} catch(e) {
  console.error(`Oops, something went wrong: ${e}`)
}
})();

```

As you can see, we created a mock version of the transaction to perform a fee calculation before removing and reading 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/4dy53oKUeh7QXr15wpKex6yXfz4xD2hMtJGdqgzvNnYyDNBZXtcgKZ7NBvCj7PCYU1ELfPZz3HEk6TzT4VQmNoS5?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:
WBAQSygkwMox2VuWKU133NxFrpDZUBdvSBeaBEue2Jq

You can find the IDL of the program here:

<https://explorer.solana.com/address/WBAQSygkwMox2VuWKU133NxFrpDZUBdvSBeaBEue2Jq?cluster=devnet>

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 typescript, we're going to go and create a type and an object for it. Let's start by creating a folder in our root directory called programs so we can easily add additional program IDLs in the future, along with a new typescript file called Turbin3_prereq.ts.

```
mkdir programs
```

```
touch ./programs/Turbin3_prereq.ts
```

Now that we've created the Turbin3_prereq.ts file, we're going to open it up and create our type and object.

```
export type Turbin3Prereq = { "version": "0.1.0", "name":  
  "Turbin3_prereq", ...etc }  
export const IDL: Turbin3Prereq = { "version": "0.1.0",  
  "name": "Turbin3_prereq", ...etc }
```

Our type and object are now ready to import this into Typescript, but to actually consume it, first, we're going to need to install Anchor, a Solana development framework, as well as define a few other imports.

Let's first install `@coral-xyz/anchor`:

```
yarn add @coral-xyz/anchor
```

Now let's open up `enroll.ts` and define the following imports:

```
import { Connection, Keypair, PublicKey } from "@solana/web3.js"
import { Program, Wallet, AnchorProvider } from "@coral-xyz/anchor"
import { IDL, Turbin3Prereq } from "../programs/Turbin3_prereq";
import wallet from "../Turbin3-wallet.json"
```

Note that we've imported a new wallet file called `Turbin3-wallet.json`. Unlike the `dev-wallet.json`, this should contain the private key for an account you might care about. To stop you from accidentally committing 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
```

As with last time, we're going to create a keypair and a connection:

```
// We're going to import our keypair from the wallet file
const keypair = Keypair.fromSecretKey(new Uint8Array(wallet));
```

```
// Create a devnet connection
```

```
const connection = new Connection("https://api.devnet.solana.com");
```

To register ourselves as having completed pre-requisites, we need to submit our github account name as a utf8 buffer:

```
// Github account
```

```
const github = Buffer.from("<your github account>", "utf8");
```

Now we're going to use our connection and wallet to create an Anchor provider:

```
// Create our anchor provider
```

```
const provider = new AnchorProvider(connection, new Wallet(keypair), {
  commitment: "confirmed"});
```

Finally, we can use the Anchor provider, our IDL object and our IDL type to create our anchor program, allowing us to interact with the Turbin3 prerequisite program.

```
// Create our program
const program : Program<Turbin3Prereq> = new Program(IDL, provider);
```

5.2 Creating a PDA

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

A Utf8 Buffer of the string: "prereq"

The Buffer of the public key of the transaction signer

There are then combined into a single Buffer, along with the program ID, to create a deterministic address for this account. The `findProgramAddressSync` function is then going to combine this with a bump to find an address that is not on the elliptic curve and return the derived address, as well as the bump which we will not be using in this example:

```
// Create the PDA for our enrollment account
const enrollment_seeds = [Buffer.from("prereq"),
  keypair.publicKey.toBuffer()];
const [enrollment_key, _bump] =
  PublicKey.findProgramAddressSync(enrollment_seeds, program.programId);
Remember to familiarize yourself with this concept as you'll be using it often!
```

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 publicKey to signify our completion of the Turbin3 pre-requisite materials!

```
// Execute our enrollment transaction
(async () => {
  try {
    const txhash = await program.methods
      .complete(github)
      .accounts({
        signer: keypair.publicKey,
      })
      .signers([
        keypair
      ]).rpc();
    console.log(`Success! Check out your TX here:
      https://explorer.solana.com/tx/${txhash}?cluster=devnet`);
  } catch(e) {
```

```
        console.error(`Oops, something went wrong: ${e}`)  
    }  
  })();
```

Congratulations, you have completed the Turbin3 Solana Pre-requisite coursework!

You are now ready for the Rust Registration Process. We will send that out at the end of the 24 hour window for the above task to all who successfully completed this one.