Prerequisites: Enrollment dApp - Typescript
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 NodeJS installed
Have yarn installed
Have a fresh folder created to follow this tutorial and all future tutorials

Now we start - you will:

- Learn how to use @solana/web3.js 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
- Create a Solana on-chain account for your Turbin3 Application
- Mint yourself a NFT that proves your successful completion of the Turbin3 Pre Reqs (Typescript)

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:
BGicGV7m2pa6WyjJnHZHFX8TDUeNCMoPRijEuyb5oQ9R

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

[228,239,246,74,243,200,55,38,173,47,72,185,164,166,211,6,227,52,245,84,58,236,14
8,176,30,89,65,194,81,97,243,166,152,155,160,181,192,90,162,54,148,145,247,96,59,
91,214,174,152,231,218,22,172,152,157,139,214,55,197,125,53,193,203,212]

If we want to save this wallet locally. To do this, we're going 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:

[228,239,246,74,243,200,55,38,173,47,72,185,164,166,211,6,227,52,245,84,58,236,14
8,176,30,89,65,194,81,97,243,166,152,155,160,181,192,90,162,54,148,145,247,96,59,
91,214,174,152,231,218,22,172,152,157,139,214,55,197,125,53,193,203,212]

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![34,46,55,124,141,190,24,204,134,91,70,184,161,181,44,122,15,172,63,62,153,150,99,255,202,89,105,77,41,89,253,130,27,195,134,14,66,75,242,7,132,234,160,203,109,195,116,251,144,44,28,56,231,114,50,131,185,168,138,61,35,98,78,53];
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/3yPVZxWb7SbR7hBN9dwWxdQswLXCKJ9cRQ4WEiHThGNwGN1m4L6cJNMMwZ8TZWE3Gn8e4FV4JziA7V14PYFMZcnf?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("87eaezi5Nou5d5MFH2DStENzWZ6iHNroDHZSbSca4RDu");
```

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 devenet. 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 your devnet wallet into your 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:

Success! Check out your TX here:
https://explorer.solana.com/tx/2tmh66GNXq2xL2bwfnh5SCmheXbArzi38VKUhEivjGXVTnTsZ9HA8AkD7brVpdU2kRBdmtwtZ4E6pB317EPQxAMy?cluster=devnet

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

You have now reached the hardest part of the admission process, we have not made things easy for you - but you got this! Read this very carefully, this is not just to follow instructions or copy paste code into your script. You will need to puzzle things out and figure out what is missing (yes there might be some things missing – Spoiler Alert: it could be one of the accounts in one of the instructions) in order to complete this challenge.

One important thing before we go deeper: you must use the Solana wallet address and Github handle that you used when you signed up for the course! You will use the

devnet tokens that you just airdropped and transferred to yourself to make some transactions with the Turbin3 enrollment dApp and confirm the completion of the pre-requisites and be one step further into your admission in the Turbin3 Builders Cohort!

Let's dive into it, starting by familiarizing 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.

And now a bigger dive into the challenge...

5.1. The Turbin3 Solana Devnet Program

There is a Turbin3 program published on the Solana devnet with a public IDL that you will need to interact to provide on-chain proof that you've made into the end of the pre-requisite 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 instructions to complete your submission:

- initialize

Receives 1 argument:
github – String

As well as 3 accounts:
user – your public key (the one you use for the Turbin3 application)
account – an account that will be created by our program with a custom PDA seed (more on this later)
system_program – the Solana system program which is responsible to create new accounts

- submit_ts

This instruction receives the following accounts:
user – your public key (the one you use for the Turbin3 application)
account – the account created by our program (see above)
mint – the address of the new asset (that will be created by our program)
collection – the collection to which the asset belongs
authority – the authority signing for the creation of the NFT (also a PDA)
mpl_core_program – the Metaplex Core program
system_program – the Solana system program

5.2. Consume an IDL in Typescript

In order for us to consume the IDL 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"
const MPL_CORE_PROGRAM_ID = new
PublicKey("CoREENxT6tW1HoK8ypY1SxRMZTcVPm7R94rH4PZNhX7d");

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
// Oops.. I forgot what I was going to write here… I guess we did this step before in this tutorial

// Create a devnet connection
// I guess this is also repeated, you should know the drill

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);

5.3. Create a PDA

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

- A Utf8 Buffer of the string: "prereqs"
- The Buffer of the public key of the transaction signer

They 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 used in this example). The code is as follows:

```
// Create the PDA for our enrollment account
const account_seeds = [
   Buffer.from("prereqs"),
   keypair.publicKey.toBuffer(),
];
const [account_key, _account_bump] =
PublicKey.findProgramAddressSync(account_seeds, program.programId);
```

Remember to familiarize yourself with this concept as you'll be using it often! You might even need to use it in other part of the enrollment script (yes this was a hint for the missing account, use it wisely! Look into the IDL that will help!)

5.4. Putting it all together

Now that we have everything we need, it's finally time to put it all together and make the
transactions interacting with the devnet program to:

- ✔ Initialize the on-chain account with the github account
- ✔ Submit the TS Prereqs and mint an NFT

All with your publicKey to signify your completion of the Turbin3 pre-requisite TS course!

First, just some missing pieces…

You need to declare the address of the mint Collection:

```
const mintCollection = new
PublicKey("5ebsp5RChCGK7ssRZMVMufgVZhd2kFbNaotcZ5UvytN2");
```

Note this address might be needed to use in some other missing part of the script (did you see this Sherlock?)

And you will need to create the mint Account for the new asset:

```
const mintTs = Keypair.generate();
```

And now the transactions…

```
// Execute the initialize transaction
(async () => {
    try {
        const txhash = await program.methods
            .initialize("github_username")
            .accountsPartial({
                user: keypair.publicKey,
                account: account_key,
                system_program: SYSTEM_PROGRAM_ID,
            })
            .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}`);
    }
})();


// Execute the submitTs transaction
(async () => {
    try {
        const txhash = await program.methods
```

```
        .submitTs()
        .accountsPartial({
            user: keypair.publicKey,
            account: account_key,
            mint: mintTs.publicKey,
            collection: mintCollection,
            mpl_core_program: MPL_CORE_PROGRAM_ID,
            system_program: SYSTEM_PROGRAM_ID,
        })
        .signers([keypair, mintTs])
        .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}`);
```

Final Notes:

- You might want to run the two methods separately (you can comment one while running the other)
- For the submitTs method you might need to change something in the IDL
- There is also something missing in the code you need to figure it out

Now, relax, if you have successfully run the two transactions… Congratulations, you have completed the Turbin3 Solana Pre-requisite Typescript 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.