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 rust 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-hat 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. You should not use AI to do this task. If we detect this, we will automatically defer to 2026.

Prerequisites:
Have NodeJS installed (v23.11 or newer)
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/kit and native crypto 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.

```shell
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/kit, along with generating a tsconfig.js configuration file.

```shell
yarn add @solana/kit @solana-program/system @types/node bs58 ws
typescript
yarn add -D tsx
touch keygen.ts
touch airdrop.ts
touch transfer.ts
touch enroll.ts
```

```shell
yarn tsc --init --rootDir ./ --outDir ./dist --esModuleInterop --lib
ES2022,DOM --module es2022 --moduleResolution bundler
--resolveJsonModule true --strict true --target ES2022
```

Finally, we're going to add the type of our project and create some scripts in our package.json file to let us run the scripts we're going to build today:

```json
{
  "name": "airdrop",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  "license": "MIT",
  "scripts": {
    "keygen": "tsx ./keygen.ts",
    "airdrop": "tsx ./airdrop.ts",
    "transfer": "tsx ./transfer.ts",
    "enroll": "tsx ./enroll.ts"
  },
  "dependencies": {
    "@solana/kit": "^3.0.3",
    "@types/node": "^24.5.2",
    "bs58": "^6.0.0",
    "typescript": "^5.9.2",
    "ws": "^8.18.3"
  },
  "devDependencies": {
    "tsx": "^4.20.5"
  }
}
```

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

1.2. Generating a Keypair

Open ./keygen.ts. We're going to generate a new keypair. We'll start by importing createKeyPairSignerFromBytes from @solana/kit

```typescript
import { createKeyPairSignerFromBytes } from "@solana/kit";
```

Now we're going to create a new Keypair, like so:

```typescript
const keypair = await crypto.subtle.generateKey(
    { name: "Ed25519" },
    true,
    ["sign", "verify"]
)
const privateKeyJwk = await crypto.subtle.exportKey('jwk',
keypair.privateKey);
const privateKeyBase64 = privateKeyJwk.d;
if (!privateKeyBase64) throw new Error('Failed to get private key
bytes')
const privateKeyBytes = new Uint8Array(Buffer.from(privateKeyBase64,
'base64'));
const publicKeyBytes = new Uint8Array(await
crypto.subtle.exportKey('raw', keypair.publicKey))
const keypairBytes = new Uint8Array([...privateKeyBytes,
...publicKeyBytes]);

const signer = await createKeyPairSignerFromBytes(keypairBytes);

console.log(`You have generated a new Solana wallet:
${signer.address}`);
```

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

```typescript
console.log(`To save your wallet, copy and paste the following into a
JSON file: [${keypairBytes}]`);
```

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

```shell
yarn keygen
```

This will generate a new Keypair, outputting its address and bytes of the keypair like so:

You've generated a new Solana wallet:
BGicGV7m2pa6WyjJnHZHFX8TDUeNCMoPRijEuyb5oQ9R

To save your wallet, copy and paste your keypair bytes 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]

We want to save this wallet locally, to do so we are going run the following command:

```
Shell
touch dev-wallet.json
```

This creates the file dev-wallet.json in our ./airdrop root directory. Now we just need to paste the keypair bytes from above into this file and save it 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 importing 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(); // gdtKSTXYULQNx87fdD3YgXkzVeyFeqwtxHm6WdEb5a9YJRnHse7GQr7t5pbepsyvU Ck7Vv 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 in another script.

We are going to open airdrop.ts and add the following imports from @solana/kit

```typescript
import { createKeyPairSignerFromBytes, createSolanaRpc,
createSolanaRpcSubscriptions, devnet, airdropFactory, lamports } from
"@solana/kit";
```

We're also going to import our wallet and recreate the Keypair object using it's bytes, while also declaring LAMPORTS_PER_SOL:

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

const LAMPORTS_PER_SOL = BigInt(1_000_000_000);

// We're going to import our keypair from the wallet file
const keypair = await createKeyPairSignerFromBytes(new
Uint8Array(wallet));
console.log(`Your Solana wallet address: ${keypair.address}`);
```

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

```typescript
// Create an rpc connection
const rpc = createSolanaRpc(devnet("https://api.devnet.solana.com"));
const rpcSubscriptions =
createSolanaRpcSubscriptions(devnet('ws://api.devnet.solana.com'));
```

Finally, we're going to airdrop 2 devnet SOL tokens to ourself using the airdropFactory function from solana kit:

```typescript
// Using the convenient airdropFactory from solana kit
const airdrop = airdropFactory({ rpc, rpcSubscriptions });

try {
    const sig = await airdrop({
        commitment: 'confirmed',
        recipientAddress: keypair.address,
        lamports: lamports(2n * LAMPORTS_PER_SOL),
    });
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${sig}?cluster=devnet`);

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

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

```shell
Success! Check out your TX here:
https://explorer.solana.com/tx/4yVMEQG8yvYqMmXezek5N6mQF6TVSktiQeK81TSde9WbxUM2rfFUq5dCQ2XZ6jWLrJUJkLbojro6pCVGSHGEeQs2?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 transfer.ts and add the following imports from @solana/kit

```typescript
import {
    address,
    appendTransactionMessageInstructions,
    assertIsTransactionWithinSizeLimit,
    compileTransaction,
    createKeyPairSignerFromBytes,
    createSolanaRpc,
    createSolanaRpcSubscriptions,
    createTransactionMessage,
    devnet,
    getSignatureFromTransaction,
    lamports,
    pipe,
    sendAndConfirmTransactionFactory,
    setTransactionMessageFeePayerSigner,
    setTransactionMessageLifetimeUsingBlockhash,
    signTransactionMessageWithSigners,
    type TransactionMessageBytesBase64
} from "@solana/kit";
```

The instruction for this transfer lives in an external library, so we have to import that as well:

```typescript
import { getTransferSolInstruction } from "@solana-program/system";
```

We will also import our dev wallet as we did last time, declare LAMPORTS_PER_SOL, and add a destination address for the transfer:

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

const LAMPORTS_PER_SOL = BigInt(1_000_000_000);

const keypair = await createKeyPairSignerFromBytes(new
Uint8Array(wallet));

// Define our Turbin3 wallet to send to
const turbin3Wallet =
address('G7MTCM2S1W6ufPhYLjodUyRZLBFbPz91CXd5C63aWoqV');
```

And create a devnet connection:

```typescript
// Create an rpc connection
const rpc = createSolanaRpc(devnet("https://api.devnet.solana.com"));
const rpcSubscriptions =
createSolanaRpcSubscriptions(devnet('ws://api.devnet.solana.com'));
```

Now we're going to create an instruction using @solana-program/system to transfer 1 SOL from our dev wallet to our Turbin3 wallet address using the provided functions from solana kit on the Solana devnet. Here's how:

```typescript
const transferInstruction = getTransferSolInstruction({
    source: keypair,
    destination: turbin3Wallet,
    amount: lamports(1n * LAMPORTS_PER_SOL)
});
const { value: latestBlockhash } = await
rpc.getLatestBlockhash().send();

const transactionMessage = pipe(
    createTransactionMessage({ version: 0 }),
    tx => setTransactionMessageFeePayerSigner(keypair, tx),
    tx => setTransactionMessageLifetimeUsingBlockhash(latestBlockhash,
tx),
    tx => appendTransactionMessageInstructions([transferInstruction],
tx)
);

const signedTransaction = await
signTransactionMessageWithSigners(transactionMessage);
```

```
assertIsTransactionWithinSizeLimit(signedTransaction);

const sendAndConfirmTransaction = sendAndConfirmTransactionFactory({
rpc, rpcSubscriptions });

try {
    await sendAndConfirmTransaction(
        signedTransaction,
        { commitment: 'confirmed' }
    );
    const signature = getSignatureFromTransaction(signedTransaction);
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signature}?cluster=devnet`);
} catch (e) {
    console.error('Transfer failed:', 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

TypeScript

```
// First get the balance from our wallet
const { value: balance } = await rpc.getBalance(keypair.address).send();

// Build a dummy transfer instruction with 0 amount to calculate the fee
const dummyTransferInstruction = getTransferSolInstruction({
    source: keypair,
    destination: turbin3Wallet,
    amount: lamports(0n)
});

const dummyTransactionMessage = pipe(
    createTransactionMessage({ version: 0 }),
    tx => setTransactionMessageFeePayerSigner(keypair, tx),
    tx => setTransactionMessageLifetimeUsingBlockhash(latestBlockhash,
tx),
    tx =>
appendTransactionMessageInstructions([dummyTransferInstruction], tx)
```

```
);

// Compile the dummy transaction message to get the message bytes
const compiledDummy = compileTransaction(dummyTransactionMessage);
const dummyMessageBase64 =
Buffer.from(compiledDummy.messageBytes).toString('base64') as
TransactionMessageBytesBase64;

// Calculate the transaction fee
const { value: fee } = await
rpc.getFeeForMessage(dummyMessageBase64).send() || 0n;

if (fee === null) {
    throw new Error('Unable to calculate transaction fee');
}

if (balance < fee ) {
    throw new Error(`Insufficient balance to cover the transaction fee.
Balance: ${balance}, Fee: ${fee}`);
}

// Calculate the exact amount to send (balance minus fee)
const sendAmount = balance - fee;

const transferInstruction = getTransferSolInstruction({
    source: keypair,
    destination: turbin3Wallet,
    amount: lamports(sendAmount)
});

const transactionMessage = pipe(
    createTransactionMessage({ version: 0 }),
    tx => setTransactionMessageFeePayerSigner(keypair, tx),
    tx => setTransactionMessageLifetimeUsingBlockhash(latestBlockhash,
tx),
    tx => appendTransactionMessageInstructions([transferInstruction],
tx)
);

const signedTransaction = await
signTransactionMessageWithSigners(transactionMessage);

assertIsTransactionWithinSizeLimit(signedTransaction);

const sendAndConfirmTransaction = sendAndConfirmTransactionFactory({
rpc, rpcSubscriptions });

try {
    await sendAndConfirmTransaction(
        signedTransaction,
        { commitment: 'confirmed' }
```

```
    );
    const signature = getSignatureFromTransaction(signedTransaction);
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signature}?cluster=devnet`);
} catch (e) {
    console.error('Transfer failed:', e);
}
```

As you can see, we created a mock version of the transaction to perform a fee calculation, then created a real transaction, signed and sent it. You can see from the transaction from the output on the block explorer, the entire value was sent to the exact lamport:

Success! Check out your TX here:
https://explorer.solana.com/tx/5qwCmVZPnGxATAgKz1iyGoq8eGYciANmZkyGo4wJBygm6q4e5Yxz3XKmEEwsTv2REnbLc5wZtMdy5PjGXsgWr4Yz?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/name that you used when you signed up for the cohort! 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 it to 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
Requires 1 argument:
github – String

Requires 3 accounts:
user - your public key (the one you use for the Turbin3 application)
account - a PDA that will be created by our program with a custom seed (more on this later)
system_program - the Solana system program which is responsible for all account creation

submit_ts
Requires 7 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 are going to use codama to create client code for the program. 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 new folders scripts and clients, and also a new file called Turbin3_prereq.json in the programs folder and generate-client.ts in the scripts folder.

```shell
mkdir programs
mkdir scripts
mkdir clients
touch ./programs/Turbin3_prereq.json
touch ./scripts/generate-client.ts
```

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

```json
{
    "address": "TRBZyQHB3m68FGeVsqTK39Wm4xejadjVhP5MAZaKWDM",
    "metadata": {
        "name": "q3_pre_reqs_rs",
        "version": "0.1.0",
         ...etc
}
```

Our IDL json is now ready to import, but to actually consume it, first, we're going to need to

install Codama, an IDL client generator, as well as define a few other imports.

Let's install codama:

```shell
yarn add codama @codama/renderers-js @codama/nodes-from-anchor
```

Next, we have to add the client generation code from the IDL json, for that we have to create a new script.

Let’s open up generate-client.ts and add the following code to generate client code.

```typescript
import { createFromRoot } from 'codama';
import { rootNodeFromAnchor, type AnchorIdl } from
'@codama/nodes-from-anchor';
import { renderVisitor as renderJavaScriptVisitor } from
"@codama/renderers-js";
import anchorIdl from '../programs/Turbin3-prereq.json'
import path from 'path';

const codama = createFromRoot(rootNodeFromAnchor(anchorIdl as
AnchorIdl));
const jsClient = path.join(import.meta.dirname, "..", "clients", "js");
codama.accept(renderJavaScriptVisitor(path.join(jsClient, "src",
"generated")));
```

Then we need to add a new script in package.json to run the client generation.

```json
 "scripts": {
    "keygen": "tsx ./keygen.ts",
    "airdrop": "tsx ./airdrop.ts",
    "transfer": "tsx ./transfer.ts",
    "enroll": "tsx ./enroll.ts",
    "generate-client": "tsx ./scripts/generate-client.ts"
  },
```

With that, we created a javascript client for the enrollment program on Devnet, based on the IDL of the program. This concept will pop up multiple times in the cohort.

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

TypeScript

```
import {
    address,
    appendTransactionMessageInstructions,
    assertIsTransactionWithinSizeLimit,
    createKeyPairSignerFromBytes,
    createSolanaRpc,
    createSolanaRpcSubscriptions,
    createTransactionMessage,
    devnet,
    getSignatureFromTransaction,
    pipe,
    sendAndConfirmTransactionFactory,
    setTransactionMessageFeePayerSigner,
    setTransactionMessageLifetimeUsingBlockhash,
    signTransactionMessageWithSigners,
    addSignersToTransactionMessage,
    getProgramDerivedAddress,
    generateKeyPairSigner,
    getAddressEncoder
} from "@solana/kit";

const MPL_CORE_PROGRAM =
address("CoREENxT6tW1HoK8ypY1SxRMZTcVPm7R94rH4PZNhX7d");
const PROGRAM_ADDRESS =
address("TRBZyQHB3m68FGeVsqTK39Wm4xejadjVhP5MAZaKWDM");
const SYSTEM_PROGRAM = address("11111111111111111111111111111111");
```

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 last time, we're going to import a keypair and execute instructions:

// 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 add some imports from the generated client code, so we can call the program correctly.

TypeScript

```
import { getInitializeInstruction, getSubmitTsInstruction } from
"./clients/js/src/generated/index";
```

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 generateProgramAddressSync 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 will not be used in this example). The code is as follows:

```
TypeScript
const addressEncoder = getAddressEncoder();
// Create the PDA for enrollment account
const accountSeeds = [Buffer.from("prereqs"),
addressEncoder.encode(keypair.address)];
const [account, _bump] = await getProgramDerivedAddress({
    programAddress: PROGRAM_ADDRESS,
    seeds: accountSeeds
});
```

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 Turbin3 dev keypair 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:

```
TypeScript
const COLLECTION =
address("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:

```typescript
// Generate mint keypair for the NFT
const mintKeyPair = await generateKeyPairSigner();
```

And now the transactions…

```typescript
// Execute the initialize transaction
const initializeIx = getInitializeInstruction({
    github: "your_github_username",
    user: keypair,
    account,
    systemProgram: SYSTEM_PROGRAM
});

// Fetch latest blockhash
const { value: latestBlockhash } = await
rpc.getLatestBlockhash().send();

const transactionMessageInit = pipe(
    createTransactionMessage({ version: 0 }),
    tx => setTransactionMessageFeePayerSigner(keypair, tx),
    tx => setTransactionMessageLifetimeUsingBlockhash(latestBlockhash,
tx),
    tx => appendTransactionMessageInstructions([initializeIx], tx)
);

const signedTxInit = await
signTransactionMessageWithSigners(transactionMessageInit);
assertIsTransactionWithinSizeLimit(signedTxInit);

const sendAndConfirmTransaction = sendAndConfirmTransactionFactory({
rpc, rpcSubscriptions });

try {
    const result = await sendAndConfirmTransaction(
        signedTxInit,
        { commitment: 'confirmed', skipPreflight: false }
    );
    console.log(result);
    const signatureInit = getSignatureFromTransaction(signedTxInit);
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signatureInit}?cluster=devnet`);
```

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

// Execute the submitTs transaction

TypeScript

```
// Execute the submitTs transaction
const submitIx = getSubmitTsInstruction({
    user: keypair,
    account,
    mint: mintKeyPair,
    collection: COLLECTION,
    authority,
    mplCoreProgram: MPL_CORE_PROGRAM,
    systemProgram: SYSTEM_PROGRAM
});

const transactionMessageSubmit = pipe(
    createTransactionMessage({ version: 0 }),
    tx => setTransactionMessageFeePayerSigner(keypair, tx),
    tx => setTransactionMessageLifetimeUsingBlockhash(latestBlockhash,
tx),
    tx => appendTransactionMessageInstructions([submitIx], tx),
    tx => addSignersToTransactionMessage([mintKeyPair], tx) // Add mint
as additional signer after appending instructions
);

const signedTxSubmit = await
signTransactionMessageWithSigners(transactionMessageSubmit);
assertIsTransactionWithinSizeLimit(signedTxSubmit);

try {
    await sendAndConfirmTransaction(
        signedTxSubmit,
        { commitment: 'confirmed', skipPreflight: false }
    );
    const signatureSubmit = getSignatureFromTransaction(signedTxSubmit);
    console.log(`Success! Check out your TX here:
https://explorer.solana.com/tx/${signatureSubmit}?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.

Part II of Typescript Assessment: (There can be no AI or discussion of this part)

In your repository for the above, create a text/ markdown file Called TS Client Logic

Read and think about the following question:

Why do you think Solana has chosen during most of its history to use Typescript for Client side development?