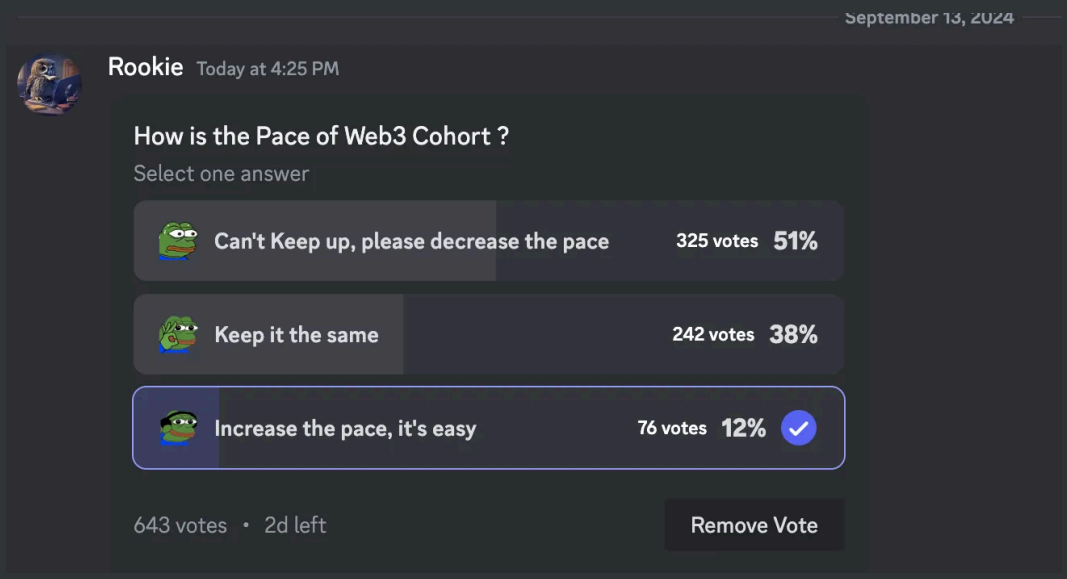
Context

Pace of the cohort

1. More offline videos, trying to keep it still 6-7 months
2. ~1 hour learning, 1 hour doubts on Fridays



Bounty from last week

$100 to Baivab. Created a token launchpad with liquidity pool creation using Raydium (this will be covered next week)

https://x.com/DuttaBaivab/status/1832117225689640991?t=f-wItD9namSPbbsqsFEEPw&s=19

Superteam hackathon

Results will be announced Monday.

Next set of offline videos -

1. Token launchpad in React (Today)
2. Wallet adapter (Tuesday)
3. Liquidity pool creation using Raydium (Thursday)

What we're learning today

1. Concept of owners
2. Owners vs authorities
3. Some common programs
4. PDAs (program derived addresses)

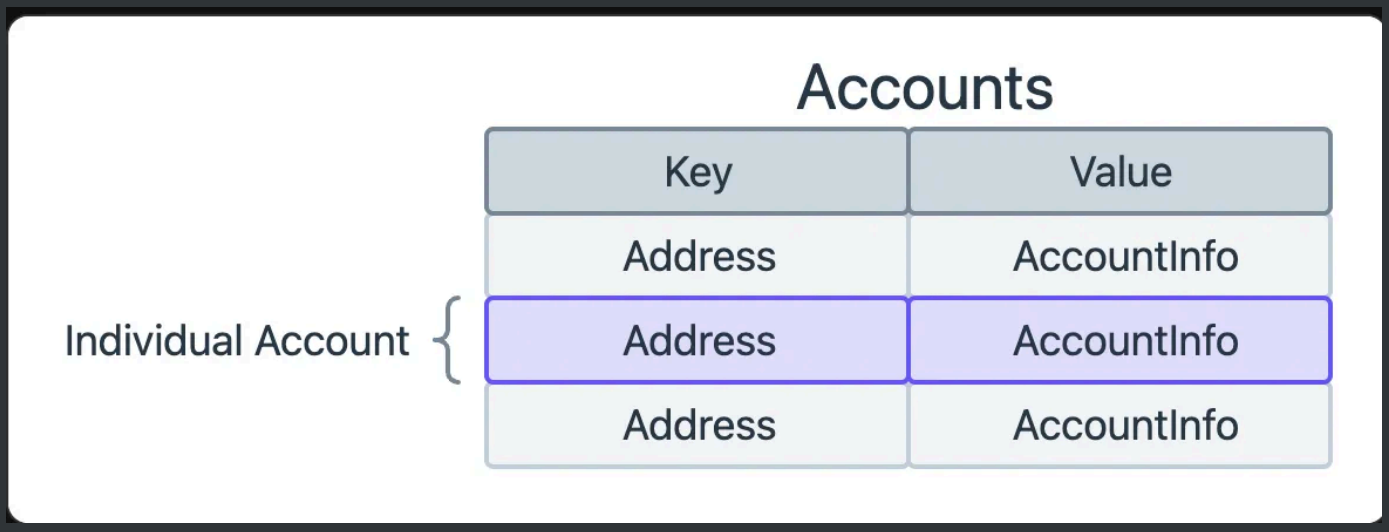
At the end of the class

GMeet to understand which amongst these feels tough to you. Only for people who are not able to follow todays class. Please dont ask questions other than the class in the gmeet

Accounts

Ref - https://solana.com/docs/core/accounts

On Solana, all data is stored in what are referred to as "accounts". The way data is organized on Solana resembles a key-value store, where each entry in the database is called an "account".



Key points

- Accounts can store up to 10MB of data, which can consist of either executable program code or program state.
  - Programs (smart contracts) are stateless accounts that store executable code.
  - Data accounts are created by programs to store and manage program state.
- Accounts require a rent deposit in SOL, proportional to the amount of data stored, which is fully refundable when the account is closed.
- Every account has a program `owner`. Only the program that owns an account can modify its data or deduct its lamport balance. However,

anyone can increase the balance.

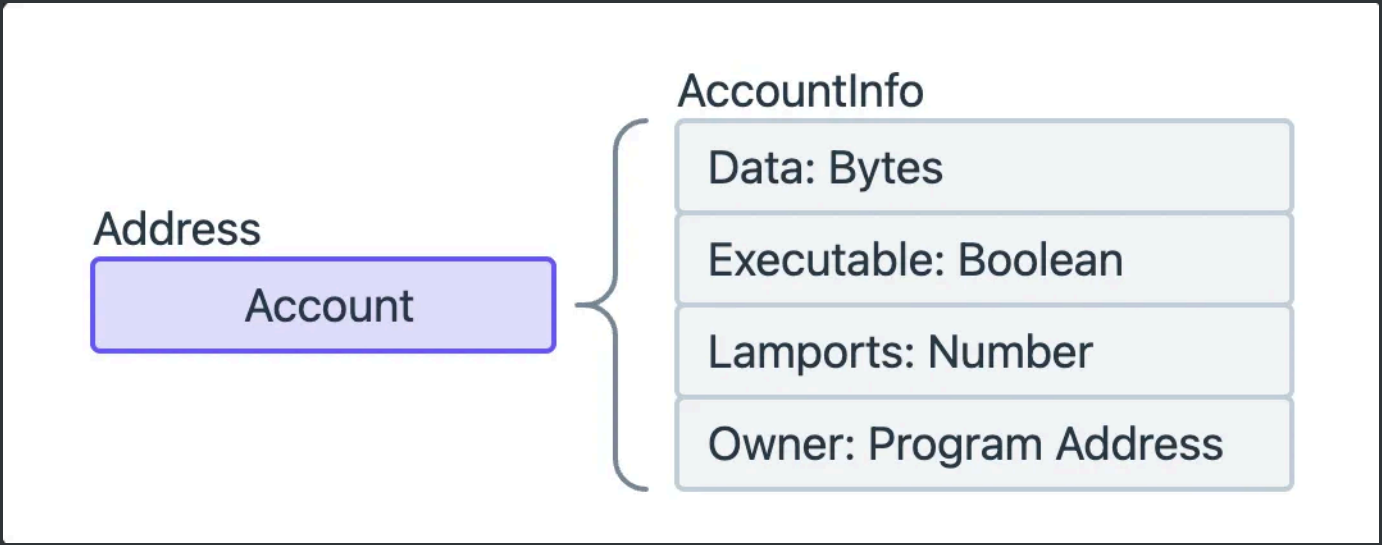
- Native programs are built-in programs included with the Solana runtime.

Account

Each account is identifiable by its unique address, represented as 32 bytes in the format of an Ed25519 `PublicKey`. You can think of the address as the unique identifier for the account.

AccountInfo

Accounts have a max size of 10MB (10 Mega Bytes) and the data stored on every account on Solana has the following structure known as the AccountInfo.



Even if you store not data, you have to store fields like `executable` and `owner` which is why you still have to have a minimum amount of SOL as rent

`solana rent 0`

Example accounts

▶ Account with no data (Owner - SystemProgram)

- Account with some data (Owner - TokenProgram)
- Program account (Owner - BPF Loader)

System program

Solana contains a small handful of native programs that are part of the validator implementation and provide various core functionalities for the network.

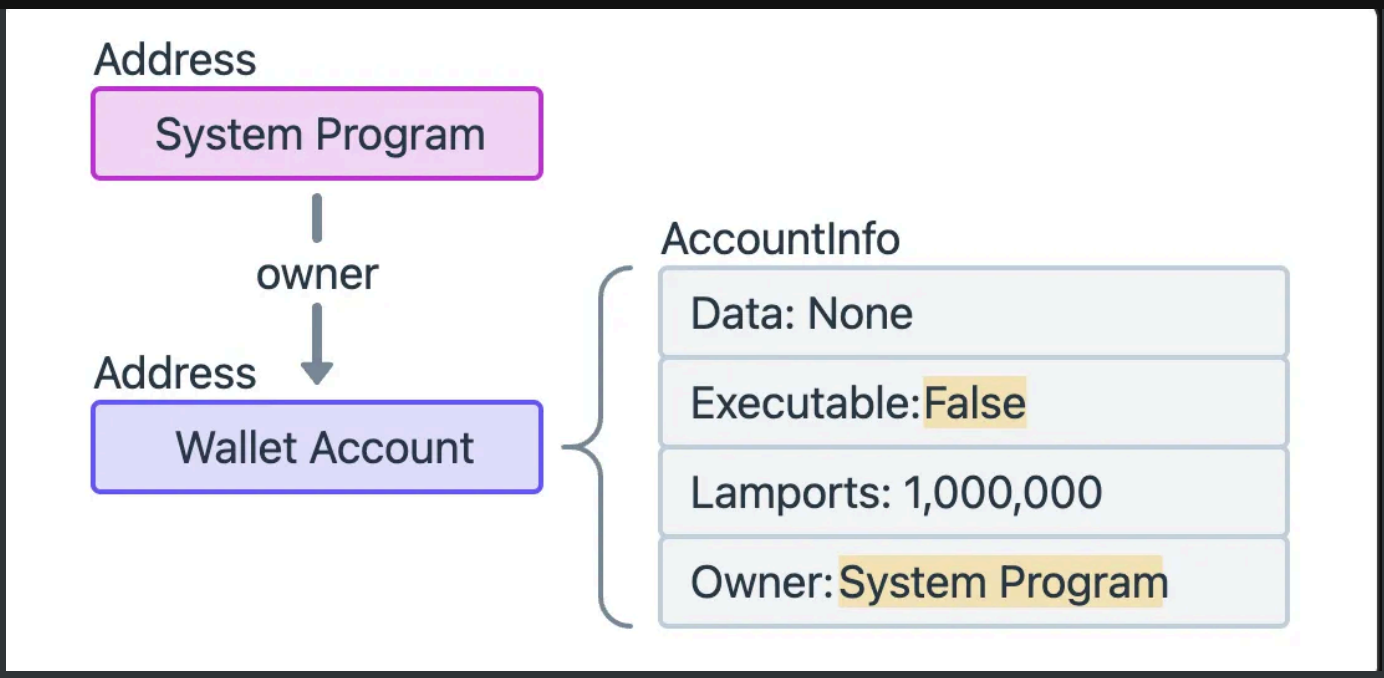
When developing custom programs on Solana, you will commonly interact with two native programs, the `System Program` and the `BPF Loader`.

System program

By default, all new accounts are owned by the System Program. The System Program performs several key tasks such as:

- New Account Creation: Only the System Program can create new accounts.
- Space Allocation: Sets the byte capacity for the data field of each account.
- Assign Program Ownership: Once the System Program creates an account, it can reassign the designated program owner to a different program account. This is how custom programs take ownership of new accounts created by the System Program.

On Solana, a `wallet` is simply an account owned by the System Program. The lamport balance of the wallet is the amount of SOL owned by the account.



Using `@solana/web3.js` to interact with the System program

- Create a new account with data and rent
- Transfer lamports from your account to another account
- Change the owner of an account

BPF Loader Program

The BPF Loader is the program designated as the "owner" of all other programs on the network, excluding Native Programs. It is responsible for deploying, upgrading, and executing custom programs.

A program I deployed just before todays class - ‣


```
"value": {
    "data": [
        "AgAAAHIUrXr9wYQNxfiS2WpxNe7z8SnPNTd8icDIJU5gUWai",
        "base64"
    ],
    "executable": true,
    "lamports": 1398960,
    "owner": "BPFLoaderUpgradeab1e11111111111111111111111",
    "rentEpoch": 18446744073709551615,
    "space": 36
}
```

Authority in solana programs

In Solana programs, `authorities` are entities or accounts that have the right to perform certain actions or make changes within the program.

For example

- Token mint authority - Can mint new tokens
- Token freeze authority - Can freeze tokens in an account
- Upgrade authority - Can `upgrade` the code of a program.

Creating and revoking mint authority

- Create a new token

```
spl-token create-token
```

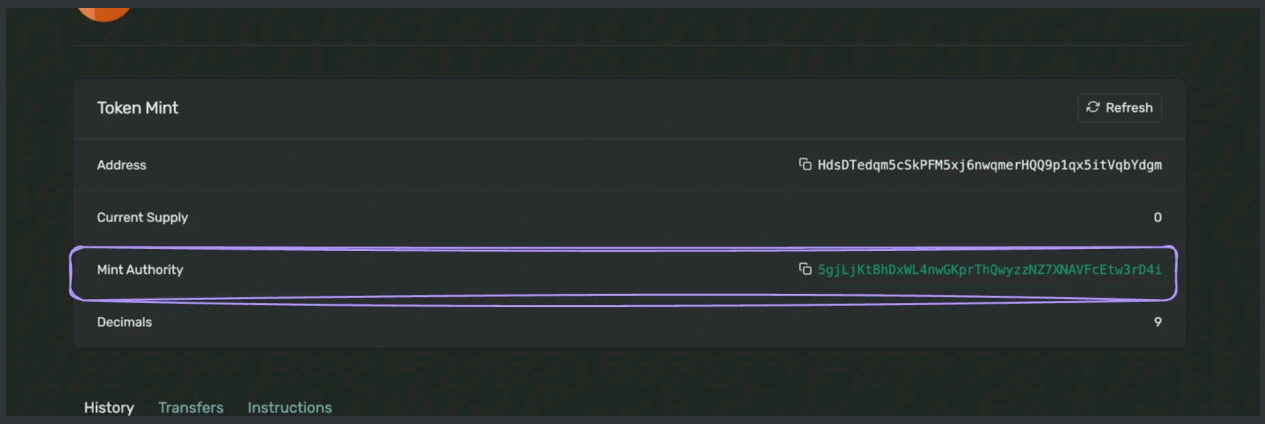
- Create an ata

```
spl-token create-account <token_mint_address>
```

- Try minting some tokens

```
spl-token mint <token_mint_address> 10000000000
```

- Check if `mint authority` exists on explorer



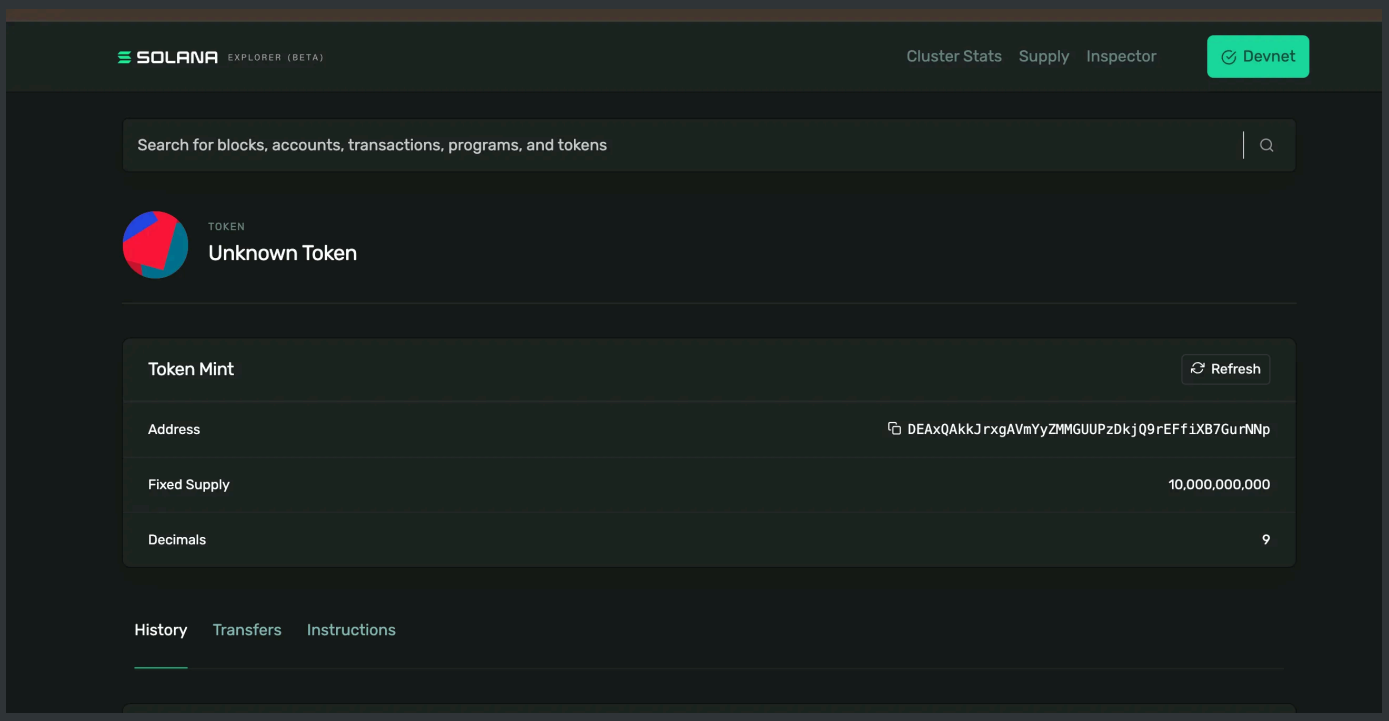
- Revoke mint authority

```
spl-token authorize <token_id> mint --disable
```

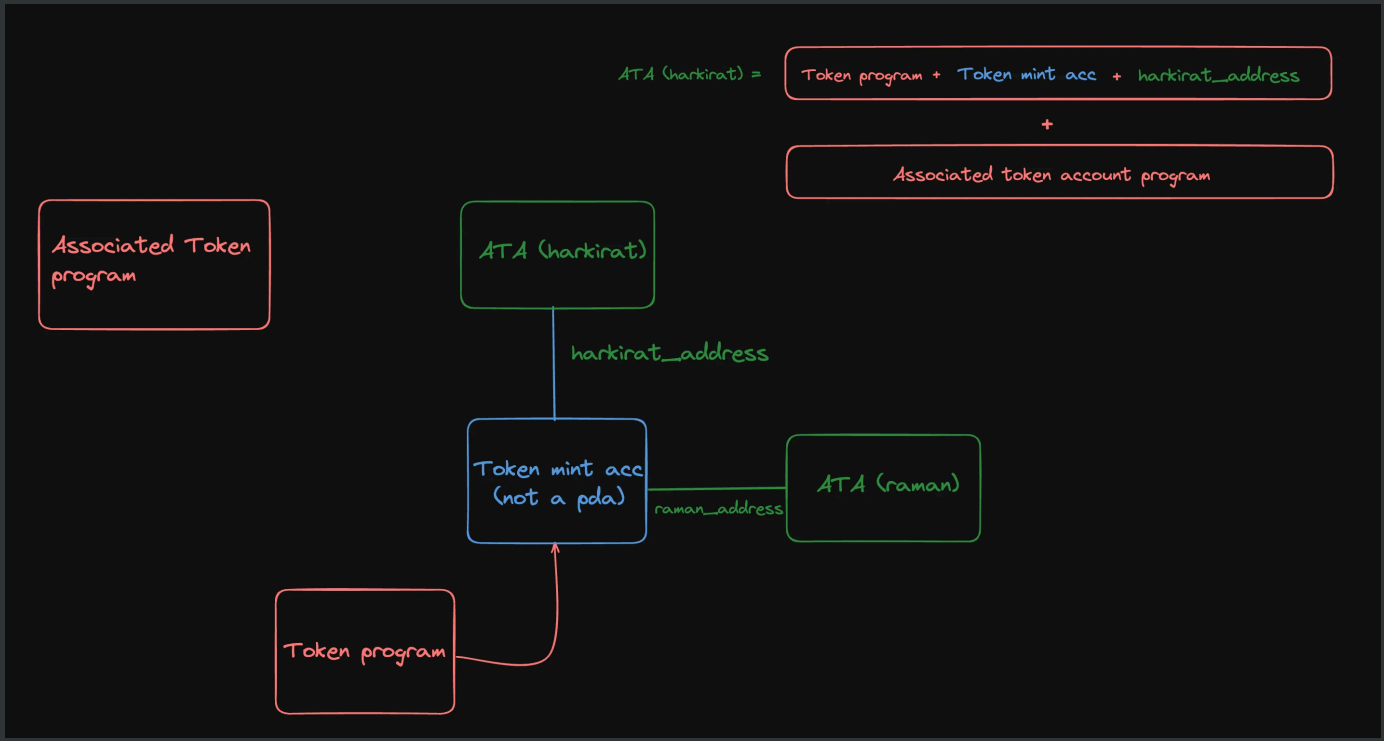
- Try to mint again/check the explorer

```
spl-token mint <token_mint_address> 10000000000
```

```
→ test3 spl-token mint DEAxQAkkJrxgAVmYyZMMGUUPzDkjQ9rEFfiXB7GurNNp 10000000000
Minting 10000000000 tokens
  Token: DEAxQAkkJrxgAVmYyZMMGUUPzDkjQ9rEFfiXB7GurNNp
  Recipient: 7xwJyp1yGHHc5ETkj88qNYMvjyV4JUhafWLv12RDVJtp
Error: Client(Error { request: Some(SendTransaction), kind: RpcError(RpcResponseError { code: -32002, message: "Transaction simulation failed: Error processing Instruction
 0: custom program error: 0x5", data: SendTransactionPreflightFailure(RpcSimulateTransactionResult { err: Some(InstructionError(0, Custom(5))), logs: Some(["Program Tokenk
egQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [1]", "Program log: Instruction: MintToChecked", "Program log: Error: the total supply of this token is fixed" "Program Token
kegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 4147 of 4147 compute units", "Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA failed: custom program error: 0x5"]), acc
ounts: None, units_consumed: Some(4147), return_data: None, inner_instructions: None }) }) })
→ test3
```



Program derived addresses



Ref - https://solana.com/docs/core/pda

Video - https://www.youtube.com/watch?v=p0eD29d8JCM

Program Derived Addresses (PDAs) provide developers on Solana with two main use cases:

- **Deterministic Account Addresses**: PDAs provide a mechanism to deterministically derive an address using a combination of optional "seeds" (predefined inputs) and a specific program ID.
- **Enable Program Signing**: The Solana runtime enables programs to "sign" for PDAs which are derived from its program ID.

Properties

- PDAs are addresses derived deterministically using
  - a combination of user-defined seeds

    - a bump seed
    - and a program's ID.
- PDAs are addresses that fall off the Ed25519 curve and have no corresponding private key.
- Solana programs can programmatically "sign" for PDAs that are derived using its program ID.
- Deriving a PDA does not automatically create an on-chain account.
- An account using a PDA as its address must be explicitly created through a dedicated instruction within a Solana program.

Find the associated token account for a user and a mint

```
const { PublicKey } = require('@solana/web3.js');
const { ASSOCIATED_TOKEN_PROGRAM_ID, TOKEN_PROGRAM_ID } = require('@

// Replace these with your actual values
const userAddress = new PublicKey('5gjLjKtBhDxWL4nwGKprThQwyzzNZ7XNAVF
const tokenMintAddress = new PublicKey('6NeR2StEEb6CP75Gsd7ydbiAkabdriM

// Derive the associated token address
const getAssociatedTokenAddress = (mintAddress, ownerAddress) => {
  return PublicKey.findProgramAddressSync(
    [
      ownerAddress.toBuffer(),
      TOKEN_PROGRAM_ID.toBuffer(),
      mintAddress.toBuffer(),
    ],
    ASSOCIATED_TOKEN_PROGRAM_ID
  );
};

const [associatedTokenAddress, bump] = getAssociatedTokenAddress(tokenM
```

```
console.log(`Associated Token Address: ${associatedTokenAddress.toBase58(
```

createProgramAddress vs findProgramAddress

```
const { PublicKey } = require('@solana/web3.js');
const { ASSOCIATED_TOKEN_PROGRAM_ID, TOKEN_PROGRAM_ID } = require('@

const PDA = PublicKey.createProgramAddressSync(
  [userAddress.toBuffer(), TOKEN_PROGRAM_ID.toBuffer(), tokenMintAddress.tol
  ASSOCIATED_TOKEN_PROGRAM_ID,
);

console.log(`PDA: ${PDA}`);
```