Code Snippets of Solana

fetchAllAirDroppedAddresses.ts

Node.js script utilizing Solana web3.js and the Helius API to fetch the airdropped addresses associated with a specific NFT collection

```
import { Connection, PublicKey } from "@solana/web3.js";
import { promises as fs } from "fs";
import path from "path";
import { URL } from "url";
import dotenv from "dotenv";
import axios from "axios";

dotenv.config();
const __dirname = new URL(".", import.meta.url).pathname;

const rpcUrl = process.env.RPC_URL;
const heliusEnrichedApiUrl = process.env.HELIUS_ENRICHED_URL;
const heliusApiKey = process.env.HELIUS_API_KEY;
if (!rpcUrl || !heliusEnrichedApiUrl || !heliusApiKey) {
  throw new Error(
    "RPC_URL or HELIUS_ENRICHED_URL or HELIUS_API_KEY is not 
  );
}

const connection = new Connection(rpcUrl!);

const getAllTransactionSignaturesForAddress = async (
  collectionMint: PublicKey
): Promise<string[]> => {
  let lastReceivedTransactionSignature = "";

  const transactionSignatures: string[] = [];

  while (true) {
    const signaturesForAddress = await connection.getSignatur
      collectionMint,
      {
        before: lastReceivedTransactionSignature
          ? lastReceivedTransactionSignature
          : undefined,
        limit: 1000,
      }
    );

    lastReceivedTransactionSignature =
      signaturesForAddress[signaturesForAddress.length - 1].s

    const signatures = signaturesForAddress.map(
      (signatureObj) => signatureObj.signature
    );

    transactionSignatures.push(...signatures);

    if (signaturesForAddress.length < 1000) {
      break;
    }
  }

  return transactionSignatures;
};

const getParsedTransactions = async (transactionSignatures: s
  const result = await axios.post(
    `${heliusEnrichedApiUrl}/v0/transactions/?api-key=${heliu
    {
      transactions: transactionSignatures,
    }
  );

  return result.data;
};

const getAirdroppedAddresses = async (collectionMint: PublicK
  const airdroppedAddresses: string[] = [];
  const transactionSignatures = await getAllTransactionSignat
    collectionMint
  );

  const batchSize = 100; // 100 is the max batch size for Hel
  for (let i = 0; i < transactionSignatures.length; i += batc
    const parsedTransactions = await getParsedTransactions(
      transactionSignatures.slice(i, i + batchSize)
    );

    // Depending on your transaction, you might need to adjus
    // This code works for cNFT minting transactions via Unde
    // To adjust this code, call the Helius API and inspect t
    const addresses = parsedTransactions.map((parsedTransacti
      const compressedEvents = parsedTransaction.events.compr

      if (compressedEvents) {
        for (let j = 0; j < compressedEvents.length; j++) {
          const compressedEvent = compressedEvents[j];

          if (compressedEvent.type === "COMPRESSED_NFT_MINT")
            return compressedEvent.newLeafOwner;
          }
        }
      }
    });

    airdroppedAddresses.push(...addresses);
  }

  return airdroppedAddresses.filter((address) => !!address);
};

const main = async () => {
  const collectionMint = new PublicKey(
    "AYED8JzJmMzq3rX61q1WsziKHtuUCLPEHK5673bWjnXr"
  );
  const airdroppedAddresses = await getAirdroppedAddresses(co

  console.log(`Found ${airdroppedAddresses.length} airdropped

  // Remove duplicate addresses
  // const uniqueAirdroppedAddresses = Array.from(new Set(air

  await fs.writeFile(
    `${path.join(
      __dirname,
      "../../output",
      `${collectionMint.toBase58()}.csv`
    )}`,
    airdroppedAddresses.join("\n")
  );
};

main();
```

fetchAllCompressedNfts.ts

Node.js script to retrieve compressed NFTs associated with a Solana Wallet Address including retries and output to a JSON file.

```
import axios from "axios";
import * as dotenv from "dotenv";
import { promises as fs } from "fs";
import path from "path";
import { URL } from "url";

dotenv.config();
const __dirname = new URL(".", import.meta.url).pathname;

const rpcUrl = process.env.RPC_URL;
if (!rpcUrl) {
  throw new Error("RPC_URL is not defined in environment vari
}
```

```
async function fetchCompressedNfts(walletAddress: string): Pr
  const body = {
    jsonrpc: '2.0',
    id: 'fetch-solana-assets',
    method: 'searchAssets',
    params: {
      ownerAddress: walletAddress,
      tokenType: 'all',
      displayOptions: {
        showNativeBalance: true,
        showInscription: true,
        showCollectionMetadata: true,
      },
    },
  };

  const maxRetries: number = 3;
  let retryCount: number = 0;
  let delayTime: number = 1000;

  while (retryCount < maxRetries) {
    try {
      const response = await axios.post(rpcUrl, body);
      if (response.data && response.data.result) {
        const items = response.data.result.items;
        const compressedNfts = items.filter((item: any) => it

        return compressedNfts;
      }
      return [];
    } catch (error: any) {
      if (error.response && error.response.status === 429) {
        console.log(`Rate limit hit, retrying after ${delayTi
        await delay(delayTime);
        retryCount++;
        delayTime *= 2;
      } else {
        console.error('Error fetching Solana assets:', error)
        throw new Error('Failed to fetch Solana assets');
      }
    }
  }
  throw new Error('Failed to fetch Solana assets after multip
}

function delay(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}

const main = async () => {
  const walletAddress = 'YourWalletAddressHere';
  try {
    const compressedNfts = await fetchCompressedNfts(walletAd
    console.log(compressedNfts);
    await fs.writeFile(
      `${path.join(__dirname, "../../output", "compressedNfts
      JSON.stringify(compressedNfts, null, 2)
    );
    console.log('Compressed NFTs data saved successfully.');
  } catch (error) {
    console.error('Failed to fetch and save compressed NFTs:'
  }
};

main();
```

fetchAllMultipleNftHolders.ts

Node.js script utilizing a DAS (Decentralized Account System) API to identify Solana addresses holding multiple NFTs from a specific collection.

```
import { PublicKey } from "@solana/web3.js";
import axios from "axios";
import { promises as fs } from "fs";
import path from "path";
import { URL } from "url";
import dotenv from "dotenv";
```

```
dotenv.config();
const __dirname = new URL(".", import.meta.url).pathname;

const dasUrl = process.env.DAS_URL;

if (!dasUrl) {
  throw new Error("DAS_URL is not defined in environment vari
}

const getNftHoldersFromCollectionMint = async (
  collectionMintAddress: PublicKey
): Promise<string[]> => {
  let page = 1;
  const holderAddresses: string[] = [];

  while (true) {
    const { data } = await axios.post(dasUrl, {
      jsonrpc: "2.0",
      id: "my-id",
      method: "getAssetsByGroup",
      params: {
        groupKey: "collection",
        groupValue: collectionMintAddress.toBase58(),
        page,
        limit: 1000,
      },
    });
    if (data.result.total === 0) break;

    data.result.items.map((asset: any) =>
      holderAddresses.push(asset.ownership.owner)
    );
    page++;
  }
  return holderAddresses;
};
```

```
const findMultipleAddresses = (addresses: string[]) => {
  const visitedAddresses: string[] = [];
  const multipleAddresses: string[] = [];

  for (let address of addresses) {
    if (
      visitedAddresses.includes(address) &&
      !multipleAddresses.includes(address)
    ) {
      multipleAddresses.push(address);
    } else {
      visitedAddresses.push(address);
    }
  }

  return multipleAddresses;
};

const main = async () => {
  const collectionMintAddress = new PublicKey(
    "J1S9H3QjnRtBbbuD4HjPV6RpRhwuk4zKbxsnCHuTgh9w"
  );

  const holderAddresses = await getNftHoldersFromCollectionMi
    collectionMintAddress
  );

  const multipleHolderAddresses = findMultipleAddresses(holde

  console.log(`Found ${multipleHolderAddresses.length} multip

  await fs.writeFile(
    `${path.join(
      __dirname,
      "../../output",
      `multiples_${collectionMintAddress.toBase58()}.csv`
    )}`,
    multipleHolderAddresses.join("\n")
  );
};

main();
```

fetchAllNftHolders.ts

Node.js script using a DAS API to retrieve and save a list of addresses holding NFTs from a specified Solana collection.

```
import { PublicKey } from "@solana/web3.js";
import axios from "axios";
import { promises as fs } from "fs";
import path from "path";
import { URL } from "url";
import dotenv from "dotenv";

dotenv.config();
const __dirname = new URL(".", import.meta.url).pathname;

const dasUrl = process.env.DAS_URL;

if (!dasUrl) {
  throw new Error("DAS_URL is not defined in environment vari
}

const getNftHoldersFromCollectionMint = async (
  collectionMintAddress: PublicKey
): Promise<string[]> => {
  let page = 1;
  const holderAddresses: string[] = [];

  while (true) {
    const { data } = await axios.post(dasUrl, {
      jsonrpc: "2.0",
      id: "my-id",
      method: "getAssetsByGroup",
      params: {
        groupKey: "collection",
        groupValue: collectionMintAddress.toBase58(),
        page,
        limit: 1000,
      },
    });
    if (data.result.total === 0) break;

    data.result.items.map((asset: any) =>
      holderAddresses.push(asset.ownership.owner)
    );
    page++;
  }
  return holderAddresses;
};

const main = async () => {
  const collectionMintAddress = new PublicKey(
    "J1S9H3QjnRtBbbuD4HjPV6RpRhwuk4zKbxsnCHuTgh9w"
  );

  const holderAddresses = await getNftHoldersFromCollectionMi
    collectionMintAddress
  );

  console.log(`Found ${holderAddresses.length} holders`);

  await fs.writeFile(
    `${path.join(
      __dirname,
      "../../output",
      `${collectionMintAddress.toBase58()}.csv`
    )}`,
    holderAddresses.join("\n")
  );
};

main();
```

fetchAllTokenHolders.ts

Node.js script to fetch and store addresses holding a specific Solana token, using direct RPC queries for efficiency.

```
import { Connection, PublicKey, type ParsedAccountData } from
import { promises as fs } from "fs";
import path from "path";
import { URL } from "url";
import dotenv from "dotenv";

dotenv.config();
const __dirname = new URL(".", import.meta.url).pathname;

const rpcUrl = process.env.RPC_URL;
if (!rpcUrl) {
  throw new Error("RPC_URL is not defined in environment vari
}

const main = async () => {
  const connection = new Connection(rpcUrl!);

  const tokenAddress = "7EYnhQoR9YM3N7UoaKRoA44Uy8JeaZV3qyouo

  const holderAddresses: string[] = [];

  const programAccounts = await connection.getParsedProgramAc
    new PublicKey("TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5D
    {
      filters: [
        {
          dataSize: 165,
        },
        {
          memcmp: {
            offset: 0,
            bytes: tokenAddress,
          },
        },
      ],
    }
  );

  for (const programAccount of programAccounts) {
    const data = programAccount.account.data as ParsedAccount
    if (data.parsed.info.tokenAmount.uiAmount > 0) {
      holderAddresses.push(data.parsed.info.owner);
      // console.log(data.parsed.info.owner);
    }
  }

  await fs.writeFile(
    `${path.join(__dirname, "../../output", `${tokenAddress}.
    holderAddresses.join("\n")
  );
};

main();
```

createSystemAccount.rs

Anchor program demonstrating how to create a new Solana system account, including logging and rent-exemption considerations.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use anchor_lang::system_program::{create_account, CreateAccou

declare_id!("ARVNCsYKDQsCLHbwUTJLpFXVrJdjhWZStyzvxmKe2xHi");

#[program]
pub mod create_system_account {
    use super::*;

    pub fn create_system_account(ctx: Context<CreateSystemAcc
        msg!("Program invoked. Creating a system account...")
        msg!(
            "  New public key will be: {}",
            &ctx.accounts.new_account.key().to_string()
        );

        // The minimum lamports for rent exemption
        let lamports = (Rent::get()?).minimum_balance(0);

        create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.new_account.to_account_i
                },
            ),
            lamports,                           // Lamports
            0,                                  // Space
            &ctx.accounts.system_program.key(), // Owner Prog
        )?;

        msg!("Account created succesfully.");
        Ok(())
    }
}

#[derive(Accounts)]
pub struct CreateSystemAccount<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(mut)]
    pub new_account: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

createAddressInfo.rs

Solana(Anchor) program to create and store structured address information.

```
#![allow(clippy::result_large_err)]
use anchor_lang::prelude::*;
use instructions::*;

pub mod instructions;
pub mod state;

declare_id!("GpVcgWdgVErgLqsn8VYUch6EqDerMgNqoLSmGyKrd6MR");/

#[program]
pub mod anchor_program_example {
    use super::*;

    pub fn create_address_info(
        ctx: Context<CreateAddressInfo>,
        name: String,
        house_number: u8,
        street: String,
        city: String,
    ) -> Result<()> {
        create::create_address_info(ctx, name, house_number,
    }
}
```

account_validation_demo.rs

Solana program showcasing Anchor's account validation mechanisms with signer checks, unchecked accounts, and ownership constraints.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

declare_id!("ECWPhR3rJbaPfyNFgphnjxSEexbTArc7vxD8fnW6tgKw");

#[program]
pub mod anchor_program_example {
    use super::*;
    pub fn check_accounts(_ctx: Context<CheckingAccounts>) ->
        Ok(())
    }
}

// Account validation in Anchor is done using the types and c
// This is a simple example and does not include all possible
#[derive(Accounts)]
pub struct CheckingAccounts<'info> {
    payer: Signer<'info>, // checks account is signer

    /// CHECK: No checks performed, example of an unchecked a
    #[account(mut)]
    account_to_create: UncheckedAccount<'info>,
    /// CHECK: Perform owner check using constraint
    #[account(
        mut,
        owner = id()
    )]
    account_to_change: UncheckedAccount<'info>,
    system_program: Program<'info, System>, // checks account
}
```

close_account_program.rs

Anchor-based Solana program providing functions to create and close user accounts, with external modules for instructions and state.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
mod instructions;
mod state;
use instructions::*;

declare_id!("99TQtoDdQ5NS2v5Ppha93aqEmv3vV9VZVfHTP5rGST3c");

#[program]
```

```
pub mod close_account_program {
    use super::*;

    pub fn create_user(ctx: Context<CreateUserContext>, name:
        create_user::create_user(ctx, name)
    }

    pub fn close_user(ctx: Context<CloseUserContext>) -> Resu
        close_user::close_user(ctx)
    }
}
```

anchor_counter.rs

Anchor program implementing a simple counter with initialization and increment functions, including account validation.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

declare_id!("BmDHboaj1kBUoinJKKSRqKfMeRKJqQqEbUj1VgzeQe4A");

#[program]
pub mod counter_anchor {
    use super::*;

    pub fn initialize_counter(_ctx: Context<InitializeCounter
        Ok(())
    }

    pub fn increment(ctx: Context<Increment>) -> Result<()> {
        ctx.accounts.counter.count = ctx.accounts.counter.cou
        Ok(())
    }
}

#[derive(Accounts)]
pub struct InitializeCounter<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(
        init,
        space = 8 + Counter::INIT_SPACE,
        payer = payer
    )]
    pub counter: Account<'info, Counter>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct Increment<'info> {
    #[account(mut)]
    pub counter: Account<'info, Counter>,
}

#[account]
#[derive(InitSpace)]
pub struct Counter {
    count: u64,
}
```

lever_interaction.rs

Anchor program demonstrating interaction with an external "Lever" program, likely to control a power status.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use lever::cpi::accounts::SetPowerStatus;
use lever::program::Lever;
use lever::{self, PowerStatus};

declare_id!("EJfTLXDCJTVwBgGpz9X2Me4CWHbvg8F8zsM7fiVJLLeR");/

#[program]
```

```
mod hand {
    use super::*;
    pub fn pull_lever(ctx: Context<PullLever>, name: String)
        // Hitting the switch_power method on the lever progr
        //
        lever::cpi::switch_power(
            CpiContext::new(
                ctx.accounts.lever_program.to_account_info(),
                // Using the accounts context struct from the
                //
                SetPowerStatus {
                    power: ctx.accounts.power.to_account_info
                },
            ),
            name,
        )
    }
}

#[derive(Accounts)]
pub struct PullLever<'info> {
    #[account(mut)]
    pub power: Account<'info, PowerStatus>,
    pub lever_program: Program<'info, Lever>,
}
```

power_switch.rs

Anchor program providing core power switching functionality, including initialization and status updates.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

declare_id!("CABVoybzrbAJSv7QhQd6GXNGKxDMRjw9niqFzizhk6uk");

#[program]
pub mod lever {
    use super::*;
    pub fn initialize(_ctx: Context<InitializeLever>) -> Resu
        Ok(())
    }

    pub fn switch_power(ctx: Context<SetPowerStatus>, name: S
        let power = &mut ctx.accounts.power;
        power.is_on = !power.is_on;

        msg!("{} is pulling the power switch!", &name);

        match power.is_on {
            true => msg!("The power is now on."),
            false => msg!("The power is now off!"),
        };

        Ok(())
    }
}

#[derive(Accounts)]
pub struct InitializeLever<'info> {
    #[account(init, payer = user, space = 8 + 8)]
    pub power: Account<'info, PowerStatus>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct SetPowerStatus<'info> {
    #[account(mut)]
    pub power: Account<'info, PowerStatus>,
}

#[account]
pub struct PowerStatus {
    pub is_on: bool,
}
```

hello_solana.rs

Basic Anchor program demonstrating how to log messages on the Solana blockchain

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

declare_id!("2phbC62wekpw95XuBk4i1KX4uA8zBUWmYbiTMhicSuBV");

#[program]
pub mod hello_solana {
    use super::*;

    pub fn hello(_ctx: Context<Hello>) -> Result<()> {
        msg!("Hello, Solana!");

        msg!("Our program's Program ID: {}", &id());

        Ok(())
    }
}

#[derive(Accounts)]
pub struct Hello {}
```

hello_solana_native.rs

Basic Solana program (without Anchor) for logging messages to the blockchain.

```
use solana_program::{
    account_info::AccountInfo, entrypoint, entrypoint::Progra
};

// Tells Solana that the entrypoint to this program
//  is the "process_instruction" function.
//
entrypoint!(process_instruction);

// Our entrypoint's parameters have to match the
//  anatomy of a transaction instruction (see README).
//
fn process_instruction(
    program_id: &Pubkey,
    _accounts: &[AccountInfo],
    _instruction_data: &[u8],
) -> ProgramResult {
    msg!("Hello, Solana!");

    msg!("Our program's Program ID: {}", &program_id);

    Ok(())
}
```

hello_solana_seahorse.rs

Basic Seahorse program demonstrating how to log messages onto Solana Blockchain

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program;
use anchor_spl::token;
use std::convert::TryFrom;

declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");

#[derive(Debug)]
#[account]
pub struct Counter {
    authority: Pubkey,
    value: u8,
}

pub fn initialize_handler(mut ctx: Context<Initialize>) -> Re
```

```
    let mut authority = &mut ctx.accounts.authority;
    let mut counter = &mut ctx.accounts.counter;
    let mut counter = counter;

    counter.authority = authority.key();

    counter.value = 0;

    msg!("{}", "Hello, Solana from Seahorse!");

    Ok(())
}

pub fn increment_handler(mut ctx: Context<Increment>) -> Resu
    let mut authority = &mut ctx.accounts.authority;
    let mut counter = &mut ctx.accounts.counter;

    counter.value += 1;

    Ok(())
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,
    #[account(
        init,
        payer = authority,
        seeds = ["new_delhi_hh".as_bytes().as_ref(), authorit
        bump,
        space = 8 + std::mem::size_of::<Counter>()
    )]
    pub counter: Box<Account<'info, Counter>>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
```

```
pub struct Increment<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,
    #[account(mut)]
    pub counter: Box<Account<'info, Counter>>,
}

#[program]
pub mod hello_solana {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>) -> Result<()>
        initialize_handler(ctx)
    }

    pub fn increment(ctx: Context<Increment>) -> Result<()> {
        increment_handler(ctx)
    }
}
```

hello_solana_solang.sol

Basic Solang prorgam demonstrating how to log messages ont Solana Blockchain

```


@program_id("F1ipperKF9EfD821ZbbYjS319LXYiBmjhzkkf5a26rC")
contract hello_solana {
    // The constructor is used to create a new account
    // Here we create a new account that stores no data and o
    @payer(payer) // The "payer" pays for the account creatio
    constructor() {
        // We get the program ID by calling 'this';
        address programId = address(this);

        // Print messages to the program logs
        print("Hello, Solana!");
        print("Our program's Program ID: {:}".format(programI
    }
}
```

pda_rent_management.rs

Anchor program enabling the creation and funding of PDA-controlled accounts to manage rent payments on Solana.

```
#![allow(clippy::result_large_err)]
use anchor_lang::prelude::*;
use instructions::*;
pub mod instructions;

declare_id!("7Hm9nsYVuBZ9rf8z9AMUHreZRv8Q4vLhqwdVTCawRZtA");

#[program]
pub mod pda_rent_payer {
    use super::*;

    pub fn init_rent_vault(ctx: Context<InitRentVault>, fund_
        init_rent_vault::init_rent_vault(ctx, fund_lamports)
    }

    pub fn create_new_account(ctx: Context<CreateNewAccount>)
        create_new_account::create_new_account(ctx)
    }
}
```

processing_instructions.rs

Simple Anchor program to process the given instructions. Below is a program simulating park attendance rules based on a visitor’s height.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

declare_id!("DgoL5J44aspizyUs9fcnpGEUJjWTLJRCfx8eYtUMYczf");

```

```
#[program]
pub mod processing_instructions {
    use super::*;

    // With Anchor, we just put instruction data in the funct
    //
    pub fn go_to_park(_ctx: Context<Park>, name: String, heig
        msg!("Welcome to the park, {}!", name);
        if height > 5 {
            msg!("You are tall enough to ride this ride. Cong
        } else {
            msg!("You are NOT tall enough to ride this ride. 
        };

        Ok(())
    }
}

#[derive(Accounts)]
pub struct Park {}
```

processing_instructions_without_anchor.rs

Simple Solana program (without Anchor) to process the given instructions. Below is a program simulating park attendance rules based on a visitor's height.

```
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::AccountInfo, entrypoint, entrypoint::Progra
};

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    _accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    // Attempt to serialize the BPF format to our struct
    //  using Borsh
    //
    let instruction_data_object = InstructionData::try_from_s

    msg!("Welcome to the park, {}!", instruction_data_object.
    if instruction_data_object.height > 5 {
        msg!("You are tall enough to ride this ride. Congratu
    } else {
        msg!("You are NOT tall enough to ride this ride. Sorr
    };

    Ok(())
}

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct InstructionData {
    name: String,
    height: u32,
}
```

page_visit_tracker.rs

Anchor program for creating and incrementing a page visit counter, utilizing external modules for state and instructions.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

use instructions::*;

pub mod instructions;
pub mod state;

declare_id!("oCCQRZyAbVxujyd8m57MPmDzZDmy2FoKW4ULS7KofCE");

#[program]
pub mod anchor_program_example {
    use super::*;
    pub fn create_page_visits(ctx: Context<CreatePageVisits>)
        create::create_page_visits(ctx)
    }

    pub fn increment_page_visits(ctx: Context<IncrementPageVi
        increment::increment_page_visits(ctx)
    }
}
```

realloc.rs

Anchor program demonstrating dynamic message reallocation, including intialization, updates, and space calculations.

```
use anchor_lang::prelude::*;

declare_id!("Fod47xKXjdHVQDzkFPBvfdWLm8gEAV4iMSXkfUzCHiSD");

#[program]
pub mod anchor_realloc {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>, input: String
        ctx.accounts.message_account.message = input;
        Ok(())
    }

    pub fn update(ctx: Context<Update>, input: String) -> Res
        ctx.accounts.message_account.message = input;
        Ok(())
    }
}

#[derive(Accounts)]
#[instruction(input: String)]
pub struct Initialize<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        init,
        payer = payer,
        space = Message::required_space(input.len()),
    )]
    pub message_account: Account<'info, Message>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(input: String)]
pub struct Update<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(
       mut,
       realloc = Message::required_space(input.len()),
       realloc::payer = payer,
       realloc::zero = true,
    )]
    pub message_account: Account<'info, Message>,
    pub system_program: Program<'info, System>,
}

#[account]
pub struct Message {
    pub message: String,
}

impl Message {
    pub fn required_space(input_len: usize) -> usize {
        8 + // 8 byte discriminator
        4 + // 4 byte for length of string
        input_len
    }
}
```

anchor_rent_calculator.rs

Anchor program demonstrating rent calculation and system account creation, including serialization for custom data.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use anchor_lang::system_program;

declare_id!("ED6f4gweAE7hWPQPXMt4kWxzDJne8VQEm9zkb1tMpFNB");

#[program]
pub mod rent_example {
    use super::*;

    pub fn create_system_account(
        ctx: Context<CreateSystemAccount>,
        address_data: AddressData,
    ) -> Result<()> {
        msg!("Program invoked. Creating a system account...")
        msg!(
            "  New public key will be: {}",
            &ctx.accounts.new_account.key().to_string()
        );

        // Determine the necessary minimum rent by calculatin
        //
        let account_span = (address_data.try_to_vec()?).len()
        let lamports_required = (Rent::get()?).minimum_balanc

        msg!("Account span: {}", &account_span);
        msg!("Lamports required: {}", &lamports_required);

        system_program::create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                system_program::CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.new_account.to_account_i
                },
            ),
            lamports_required,
            account_span as u64,
            &ctx.accounts.system_program.key(),
        )?;

        msg!("Account created succesfully.");
        Ok(())
    }
}

#[derive(Accounts)]
pub struct CreateSystemAccount<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(mut)]
    pub new_account: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(AnchorSerialize, AnchorDeserialize, Debug)]
pub struct AddressData {
    name: String,
    address: String,
}
```

carnival_activities.rs

A simple Anchor program simulating carnival activities (rides, games, food) with instructions and modular design.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

pub mod error;
pub mod instructions;
pub mod state;

use crate::instructions::{eat_food, get_on_ride, play_game};

// For setting up modules & configs

declare_id!("8t94SEJh9jVjDwV7cbiuT6BvEsHo4YHP9x9a5rYH1NpP");

#[program]
pub mod carnival {
    use super::*;

    pub fn go_on_ride(
        _ctx: Context<CarnivalContext>,
        name: String,
        height: u32,
        ticket_count: u32,
        ride_name: String,
    ) -> Result<()> {
        get_on_ride::get_on_ride(get_on_ride::GetOnRideInstru
            rider_name: name,
            rider_height: height,
            rider_ticket_count: ticket_count,
            ride: ride_name,
        })
    }

    pub fn play_game(
        _ctx: Context<CarnivalContext>,
        name: String,
        ticket_count: u32,
        game_name: String,
    ) -> Result<()> {
        play_game::play_game(play_game::PlayGameInstructionDa
            gamer_name: name,
            gamer_ticket_count: ticket_count,
            game: game_name,
        })
    }

    pub fn eat_food(
        _ctx: Context<CarnivalContext>,
        name: String,
        ticket_count: u32,
        food_stand_name: String,
    ) -> Result<()> {
        eat_food::eat_food(eat_food::EatFoodInstructionData {
            eater_name: name,
            eater_ticket_count: ticket_count,
            food_stand: food_stand_name,
        })
    }
}

#[derive(Accounts)]
pub struct CarnivalContext<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
}
```

transfer_sol.rs

Anchor program demonstrating both CPI and direct methods for transferring SOL, including account checks.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use anchor_lang::system_program;

declare_id!("4fQVnLWKKKYxtxgGn7Haw8v2g2Hzbu8K61JvWKvqAi7W");

#[program]
pub mod transfer_sol {
    use super::*;

    pub fn transfer_sol_with_cpi(ctx: Context<TransferSolWith
        system_program::transfer(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                system_program::Transfer {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.recipient.to_account_inf
                },
            ),
            amount,
        )?;

        Ok(())
    }

    // Directly modifying lamports is only possible if the pr
    pub fn transfer_sol_with_program(
        ctx: Context<TransferSolWithProgram>,
        amount: u64,
    ) -> Result<()> {
        **ctx.accounts.payer.try_borrow_mut_lamports()? -= am
        **ctx.accounts.recipient.try_borrow_mut_lamports()? +
        Ok(())
    }
}

#[derive(Accounts)]
pub struct TransferSolWithCpi<'info> {
    #[account(mut)]
    payer: Signer<'info>,
    #[account(mut)]
    recipient: SystemAccount<'info>,
    system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct TransferSolWithProgram<'info> {
    /// CHECK: Use owner constraint to check account is owned
    #[account(
        mut,
        owner = id() // value of declare_id!()
    )]
    payer: UncheckedAccount<'info>,
    #[account(mut)]
    recipient: SystemAccount<'info>,
}
```

transfer_sol_seahorse.rs

A simple Seahorse program for transferring SOL.

```
#![allow(unused_imports)]
#![allow(unused_variables)]
#![allow(unused_mut)]

pub mod dot;

use anchor_lang::prelude::*;
use anchor_spl::{
    associated_token::{self, AssociatedToken},
    token::{self, Mint, Token, TokenAccount},
};

use dot::program::*;
use std::{cell::RefCell, rc::Rc};

declare_id!("2RjL4mpTANyGxz7fLWEbQtmdEDti7c4CqsLR96mgvcaV");

pub mod seahorse_util {
    use super::*;

    #[cfg(feature = "pyth-sdk-solana")]
    pub use pyth_sdk_solana::{load_price_feed_from_account_in
    use std::{collections::HashMap, fmt::Debug, ops::Deref};

    pub struct Mutable<T>(Rc<RefCell<T>>);

    impl<T> Mutable<T> {
        pub fn new(obj: T) -> Self {
            Self(Rc::new(RefCell::new(obj)))
        }
    }

    impl<T> Clone for Mutable<T> {
        fn clone(&self) -> Self {
            Self(self.0.clone())
        }
    }

    impl<T> Deref for Mutable<T> {
        type Target = Rc<RefCell<T>>;

        fn deref(&self) -> &Self::Target {
            &self.0
        }
    }

    impl<T: Debug> Debug for Mutable<T> {
        fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std
            write!(f, "{:?}", self.0)
        }
    }

    impl<T: Default> Default for Mutable<T> {
        fn default() -> Self {
            Self::new(T::default())
        }
    }

    impl<T: Clone> Mutable<Vec<T>> {
        pub fn wrapped_index(&self, mut index: i128) -> usize
            if index >= 0 {
                return index.try_into().unwrap();
            }

            index += self.borrow().len() as i128;
            return index.try_into().unwrap();
        }
    }

    impl<T: Clone, const N: usize> Mutable<[T; N]> {
        pub fn wrapped_index(&self, mut index: i128) -> usize
            if index >= 0 {
                return index.try_into().unwrap();
            }

            index += self.borrow().len() as i128;

            return index.try_into().unwrap();
        }
    }

    #[derive(Clone)]
    pub struct Empty<T: Clone> {
        pub account: T,
        pub bump: Option<u8>,
    }

    #[derive(Clone, Debug)]
    pub struct ProgramsMap<'info>(pub HashMap<&'static str, A

    impl<'info> ProgramsMap<'info> {
        pub fn get(&self, name: &'static str) -> AccountInfo<
            self.0.get(name).unwrap().clone()
        }
    }

    #[derive(Clone, Debug)]
    pub struct WithPrograms<'info, 'entrypoint, A> {
        pub account: &'entrypoint A,
        pub programs: &'entrypoint ProgramsMap<'info>,
    }
```

```
impl<'info, 'entrypoint, A> Deref for WithPrograms<'info,
    type Target = A;

    fn deref(&self) -> &Self::Target {
        &self.account
    }
}

pub type SeahorseAccount<'info, 'entrypoint, A> =
    WithPrograms<'info, 'entrypoint, Box<Account<'info, A

pub type SeahorseSigner<'info, 'entrypoint> = WithProgram

#[derive(Clone, Debug)]
pub struct CpiAccount<'info> {
    #[doc = "CHECK: CpiAccounts temporarily store Account
    pub account_info: AccountInfo<'info>,
    pub is_writable: bool,
    pub is_signer: bool,
    pub seeds: Option<Vec<Vec<u8>>>,
}

#[macro_export]
macro_rules! seahorse_const {
    ($ name : ident , $ value : expr) => {
        macro_rules! $name {
            () => {
                $value
            };
        }

        pub(crate) use $name;
    };
}

#[macro_export]
macro_rules! assign {
    ($ lval : expr , $ rval : expr) => {{
            let temp = $rval;

            $lval = temp;
        }};
    }

    #[macro_export]
    macro_rules! index_assign {
        ($ lval : expr , $ idx : expr , $ rval : expr) => {
            let temp_rval = $rval;
            let temp_idx = $idx;

            $lval[temp_idx] = temp_rval;
        };
    }

    pub(crate) use assign;

    pub(crate) use index_assign;

    pub(crate) use seahorse_const;
}

#[program]
mod seahorse {
    use super::*;
    use seahorse_util::*;
    use std::collections::HashMap;

    #[derive(Accounts)]
    pub struct InitMockAccount<'info> {
        #[account(mut)]
        pub signer: Signer<'info>,
        # [account (init , space = std :: mem :: size_of :: <
        pub mock_account: Box<Account<'info, dot::program::Mo
        pub rent: Sysvar<'info, Rent>,
        pub system_program: Program<'info, System>,
    }
```

```
pub fn init_mock_account(ctx: Context<InitMockAccount>) -:
    let mut programs = HashMap::new();

    programs.insert(
        "system_program",
        ctx.accounts.system_program.to_account_info(),
    );

    let programs_map = ProgramsMap(programs);
    let signer = SeahorseSigner {
        account: &ctx.accounts.signer,
        programs: &programs_map,
    };

    let mock_account = Empty {
        account: dot::program::MockAccount::load(&mut ctx
        bump: ctx.bumps.get("mock_account").map(|bump| *b
    };

    init_mock_account_handler(signer.clone(), mock_accoun

    dot::program::MockAccount::store(mock_account.account

    return Ok(());
}

#[derive(Accounts)]
# [instruction (amount : u64)]
pub struct TransferSolWithCpi<'info> {
    #[account(mut)]
    pub sender: Signer<'info>,
    #[account(mut)]
    pub recipient: Box<Account<'info, dot::program::MockA
    pub system_program: Program<'info, System>,
}

pub fn transfer_sol_with_cpi(ctx: Context<TransferSolWith
        let mut programs = HashMap::new();

        programs.insert(
            "system_program",
            ctx.accounts.system_program.to_account_info(),
        );

        let programs_map = ProgramsMap(programs);
        let sender = SeahorseSigner {
            account: &ctx.accounts.sender,
            programs: &programs_map,
        };

        let recipient = dot::program::MockAccount::load(&mut 

        transfer_sol_with_cpi_handler(sender.clone(), recipie

        dot::program::MockAccount::store(recipient);

        return Ok(());
    }
}
```

cnft_burn.rs

Anchor program enabling cNFT (compressed NFT) burning via Bubblegum program, including Merkle tree validation and compression program integration.

```
use anchor_lang::prelude::*;

declare_id!("FbeHkUEevbhKmdk5FE5orcTaJkCYn5drwZoZXaxQXXNn");

#[derive(Clone)]
pub struct SPLCompression;

impl anchor_lang::Id for SPLCompression {
    fn id() -> Pubkey {
        spl_account_compression::id()
    }
```

```
}

#[program]
pub mod cnft_burn {
    use super::*;

    pub fn burn_cnft<'info>(
        ctx: Context<'_, '_, '_, 'info, BurnCnft<'info>>,
        root: [u8; 32],
        data_hash: [u8; 32],
        creator_hash: [u8; 32],
        nonce: u64,
        index: u32,
    ) -> Result<()> {
        let tree_config = ctx.accounts.tree_authority.to_acco
        let leaf_owner = ctx.accounts.leaf_owner.to_account_i
        let merkle_tree = ctx.accounts.merkle_tree.to_account_
        let log_wrapper = ctx.accounts.log_wrapper.to_account_
        let compression_program = ctx.accounts.compression_pr
        let system_program = ctx.accounts.system_program.to_a

        let cnft_burn_cpi = mpl_bubblegum::instructions::Burn
            &ctx.accounts.bubblegum_program,
            mpl_bubblegum::instructions::BurnCpiAccounts {
                tree_config: &tree_config,
                leaf_owner: (&leaf_owner, true),
                leaf_delegate: (&leaf_owner, false),
                merkle_tree: &merkle_tree,
                log_wrapper: &log_wrapper,
                compression_program: &compression_program,
                system_program: &system_program,
            },
            mpl_bubblegum::instructions::BurnInstructionArgs
                root,
                data_hash,
                creator_hash,
                nonce,
                index,
            },
        );

        cnft_burn_cpi.invoke_with_remaining_accounts(
            ctx.remaining_accounts
                .iter()
                .map(|account| (account, false, false))
                .collect::<Vec<_>>()
                .as_slice(),
        )?;

        Ok(())
    }
}

#[derive(Accounts)]
pub struct BurnCnft<'info> {
    #[account(mut)]
    pub leaf_owner: Signer<'info>,
    #[account(mut)]
    #[account(
        seeds = [merkle_tree.key().as_ref()],
        bump,
        seeds::program = bubblegum_program.key()
    )]
    /// CHECK: This account is modified in the downstream pro
    pub tree_authority: UncheckedAccount<'info>,
    #[account(mut)]
    /// CHECK: This account is neither written to nor read fr
    pub merkle_tree: UncheckedAccount<'info>,
    /// CHECK: This account is neither written to nor read fr
    pub log_wrapper: UncheckedAccount<'info>,
    pub compression_program: Program<'info, SPLCompression>,
    /// CHECK: This account is neither written to nor read fr
    pub bubblegum_program: UncheckedAccount<'info>,
    pub system_program: Program<'info, System>,
}
```

compressed_nft_mint.rs

Solidity Solana contract enabling compressed NFT minting via interaction with the Metaplex Bubblegum program.

```
import "solana";

@program_id("BvgEJTPXfriGPopjJr1nLc4vADXm7A7TqjLFVztpd19Q")
contract compressed_nft {

    @payer(payer) // payer address
    @seed("seed") // hardcoded seed
    constructor(
        @bump bytes1 bump // bump seed for pda address
    ) {
        // Creating a dataAccount for the program, which is r
        // However, this account is not used in the program
    }

    // Mint a compressed NFT to an existing merkle tree, via 
    // Reference: https://github.com/metaplex-foundation/meta
    // Reference: https://github.com/metaplex-foundation/meta
    @mutableAccount(tree_authority) // authority of the merkl
    @account(leaf_owner) // owner of the new compressed NFT
    @account(leaf_delegate) // delegate of the new compressed
    @mutableAccount(merkle_tree)  // address of the merkle tr
    @mutableSigner(payer) // payer
    @mutableSigner(tree_delegate) // delegate of the merkle t
    @account(noop_address)
    @account(compression_pid)
    @account(bubblegum_pid)
    function mint(
        string uri // uri of the new compressed NFT (metadata
    ) external {
        print("Minting Compressed NFT");

        // Create a creator array with a single creator
        Creator[] memory creators = new Creator[](1);
        // Set the creator to the payer
        creators[0] = Creator({
            creatorAddress: tx.accounts.payer.key,
            verified: false,
            share: 100
        });

        // Create the metadata args, representing the metadat
        // Solidity does not support optional arguments,
        // So we have to explicitly declare if the optional a
        // If not present, we comment them out, otherwise the
        MetadataArgs memory args = MetadataArgs({
            name: "RGB",
            symbol: "RGB",
            uri: uri,
            sellerFeeBasisPoints: 0,
            primarySaleHappened: false,
            isMutable: true,
            editionNoncePresent: false,
            // editionNonce: 0,
            tokenStandardPresent: true,
            tokenStandard: TokenStandard.NonFungible,
            collectionPresent: false,
            // collection: Collection({
            //     verified: false,
            //     key: address(0)
            // }),
            usesPresent: false,
            // uses: Uses({
            //     useMethod: UseMethod.Burn,
            //     remaining: 0,
            //     total: 0
            // }),
            tokenProgramVersion: TokenProgramVersion.Original
            creators: creators
        });

        AccountMeta[9] metas = [
            AccountMeta({pubkey: tx.accounts.tree_authority.k
            AccountMeta({pubkey: tx.accounts.leaf_owner.key, 
            AccountMeta({pubkey: tx.accounts.leaf_delegate.ke
            AccountMeta({pubkey: tx.accounts.merkle_tree.key,
            AccountMeta({pubkey: tx.accounts.payer.key, is_wr
            AccountMeta({pubkey: tx.accounts.tree_delegate.ke
            AccountMeta({pubkey: tx.accounts.noop_address.key
            AccountMeta({pubkey: tx.accounts.compression_pid.
            AccountMeta({pubkey: address"11111111111111111111
        ];

        // Reference: https://github.com/metaplex-foundation/
        bytes8 discriminator = 0x9162c076b8937668;
        bytes instructionData = abi.encode(discriminator, arg

        // Invoking the Bubblegum program
        tx.accounts.bubblegum_pid.key.call{accounts: metas}(i
    }

    // Reference: https://github.com/metaplex-foundation/meta
    struct MetadataArgs {
        string name;
        string symbol;
        string uri;
        uint16 sellerFeeBasisPoints;
        bool primarySaleHappened;
        bool isMutable;
        bool editionNoncePresent;
        // uint8 editionNonce;
        bool tokenStandardPresent;
        TokenStandard tokenStandard;
        bool collectionPresent;
        // Collection collection;
        bool usesPresent;
        // Uses uses;
        TokenProgramVersion tokenProgramVersion;
        Creator[] creators;
    }
```

```
}
```

```
    enum TokenStandard {
        NonFungible,
        FungibleAsset,
        Fungible,
        NonFungibleEdition
    }

    enum TokenProgramVersion {
        Original,
        Token2022
    }

    struct Creator {
        address creatorAddress;
        bool verified;
        uint8 share;
    }

    struct Collection {
        bool verified;
        address key;
    }

    struct Uses {
        UseMethod useMethod;
        uint64 remaining;
        uint64 total;
    }

    enum UseMethod {
        Burn,
        Multiple,
        Single
    }

}
```

cnft_vault_withdraw.rs

Solana program (written with a mix of Anchor and manual CPI) enabling withdrawal of compressed NFTs from vaults, interacting with the Metaplex Bubblegum program.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use mpl_bubblegum::state::TreeConfig;
use solana_program::pubkey::Pubkey;
use spl_account_compression::{program::SplAccountCompression,

declare_id!("CNftyK7T8udPwYRzZUMWzbh79rKrz9a5GwV2wv7iEHpk");

#[derive(Clone)]
pub struct MplBubblegum;

impl anchor_lang::Id for MplBubblegum {
    fn id() -> Pubkey {
        mpl_bubblegum::id()
    }
}

// first 8 bytes of SHA256("global:transfer")
const TRANSFER_DISCRIMINATOR: &[u8; 8] = &[163, 52, 200, 231,

#[program]
pub mod cnft_vault {

    use super::*;

    pub fn withdraw_cnft<'info>(
        ctx: Context<'_, '_, '_, 'info, Withdraw<'info>>,
        root: [u8; 32],
        data_hash: [u8; 32],
        creator_hash: [u8; 32],
        nonce: u64,
        index: u32,
    ) -> Result<()> {
        msg!(
            "attempting to send nft {} from tree {}",
            index,
            ctx.accounts.merkle_tree.key()
        );

        let mut accounts: Vec<solana_program::instruction::Ac
            AccountMeta::new_readonly(ctx.accounts.tree_autho
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.new_leaf_o
            AccountMeta::new(ctx.accounts.merkle_tree.key(), 
            AccountMeta::new_readonly(ctx.accounts.log_wrappe
            AccountMeta::new_readonly(ctx.accounts.compressio
            AccountMeta::new_readonly(ctx.accounts.system_pro
        ];

        let mut data: Vec<u8> = vec![];
        data.extend(TRANSFER_DISCRIMINATOR);
        data.extend(root);
        data.extend(data_hash);
        data.extend(creator_hash);
        data.extend(nonce.to_le_bytes());
        data.extend(index.to_le_bytes());

        let mut account_infos: Vec<AccountInfo> = vec![
            ctx.accounts.tree_authority.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.new_leaf_owner.to_account_info(),
            ctx.accounts.merkle_tree.to_account_info(),
            ctx.accounts.log_wrapper.to_account_info(),
            ctx.accounts.compression_program.to_account_info(
            ctx.accounts.system_program.to_account_info(),
        ];

        // add "accounts" (hashes) that make up the merkle pr
        for acc in ctx.remaining_accounts.iter() {
            accounts.push(AccountMeta::new_readonly(acc.key()
            account_infos.push(acc.to_account_info());
        }

        msg!("manual cpi call");
        solana_program::program::invoke_signed(
            &solana_program::instruction::Instruction {
                program_id: ctx.accounts.bubblegum_program.ke
                accounts,
                data,
            },
            &account_infos[..],
            &[&[b"cNFT-vault", &[*ctx.bumps.get("leaf_owner")
        )
        .map_err(Into::into)
    }

    #[allow(clippy::too_many_arguments)]
    pub fn withdraw_two_cnfts<'info>(
        ctx: Context<'_, '_, '_, 'info, WithdrawTwo<'info>>,
        root1: [u8; 32],
        data_hash1: [u8; 32],
        creator_hash1: [u8; 32],
        nonce1: u64,
        index1: u32,
        proof_1_length: u8,
        root2: [u8; 32],
        data_hash2: [u8; 32],
        creator_hash2: [u8; 32],
        nonce2: u64,
        index2: u32,
        _proof_2_length: u8, // we don't actually need this (
    ) -> Result<()> {
        let merkle_tree1 = ctx.accounts.merkle_tree1.key();
        let merkle_tree2 = ctx.accounts.merkle_tree2.key();
        msg!(
            "attempting to send nfts from trees {} and {}",
            merkle_tree1,
            merkle_tree2
        );

        // Note: in this example anyone can withdraw any NFT 
        // in productions you should check if nft transfers a

        let mut accounts1: Vec<solana_program::instruction::A
            AccountMeta::new_readonly(ctx.accounts.tree_autho
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.new_leaf_o
            AccountMeta::new(ctx.accounts.merkle_tree1.key(),
            AccountMeta::new_readonly(ctx.accounts.log_wrappe
            AccountMeta::new_readonly(ctx.accounts.compressio
            AccountMeta::new_readonly(ctx.accounts.system_pro
        ];

        let mut accounts2: Vec<solana_program::instruction::A
            AccountMeta::new_readonly(ctx.accounts.tree_autho
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.leaf_owner
            AccountMeta::new_readonly(ctx.accounts.new_leaf_o
            AccountMeta::new(ctx.accounts.merkle_tree2.key(),
            AccountMeta::new_readonly(ctx.accounts.log_wrappe
            AccountMeta::new_readonly(ctx.accounts.compressio
            AccountMeta::new_readonly(ctx.accounts.system_pro
        ];

        let mut data1: Vec<u8> = vec![];
        data1.extend(TRANSFER_DISCRIMINATOR);
        data1.extend(root1);
        data1.extend(data_hash1);
        data1.extend(creator_hash1);
        data1.extend(nonce1.to_le_bytes());
        data1.extend(index1.to_le_bytes());
        let mut data2: Vec<u8> = vec![];
        data2.extend(TRANSFER_DISCRIMINATOR);
        data2.extend(root2);
        data2.extend(data_hash2);
        data2.extend(creator_hash2);
        data2.extend(nonce2.to_le_bytes());
        data2.extend(index2.to_le_bytes());

        let mut account_infos1: Vec<AccountInfo> = vec![
            ctx.accounts.tree_authority1.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.new_leaf_owner1.to_account_info(),
            ctx.accounts.merkle_tree1.to_account_info(),
            ctx.accounts.log_wrapper.to_account_info(),
            ctx.accounts.compression_program.to_account_info(
            ctx.accounts.system_program.to_account_info(),
        ];
        let mut account_infos2: Vec<AccountInfo> = vec![
            ctx.accounts.tree_authority2.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.leaf_owner.to_account_info(),
            ctx.accounts.new_leaf_owner2.to_account_info(),
            ctx.accounts.merkle_tree2.to_account_info(),
            ctx.accounts.log_wrapper.to_account_info(),
            ctx.accounts.compression_program.to_account_info(
            ctx.accounts.system_program.to_account_info(),
        ];

        for (i, acc) in ctx.remaining_accounts.iter().enumera
            if i < proof_1_length as usize {
                accounts1.push(AccountMeta::new_readonly(acc.
                account_infos1.push(acc.to_account_info());
            } else {
                accounts2.push(AccountMeta::new_readonly(acc.
                account_infos2.push(acc.to_account_info());
            }
        }

        msg!("withdrawing cNFT#1");
        solana_program::program::invoke_signed(
            &solana_program::instruction::Instruction {
                program_id: ctx.accounts.bubblegum_program.ke
                accounts: accounts1,
                data: data1,
            },
            &account_infos1[..],
            &[&[b"cNFT-vault", &[*ctx.bumps.get("leaf_owner")
        )?;

        msg!("withdrawing cNFT#2");
        solana_program::program::invoke_signed(
            &solana_program::instruction::Instruction {
                program_id: ctx.accounts.bubblegum_program.ke
                accounts: accounts2,
                data: data2,
            },
            &account_infos2[..],
            &[&[b"cNFT-vault", &[*ctx.bumps.get("leaf_owner")
        )?;

        msg!("successfully sent cNFTs");
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Withdraw<'info> {
    #[account(
        seeds = [merkle_tree.key().as_ref()],
        bump,
        seeds::program = bubblegum_program.key()
    )]
    /// CHECK: This account is neither written to nor read fr
    pub tree_authority: Account<'info, TreeConfig>,

    #[account(
        seeds = [b"cNFT-vault"],
        bump,
    )]
    /// CHECK: This account doesnt even exist (it is just the
    pub leaf_owner: UncheckedAccount<'info>, // sender (the v
    /// CHECK: This account is neither written to nor read fr
    pub new_leaf_owner: UncheckedAccount<'info>, // receiver
    #[account(mut)]
    /// CHECK: This account is modified in the downstream pro
    pub merkle_tree: UncheckedAccount<'info>,
    pub log_wrapper: Program<'info, Noop>,
    pub compression_program: Program<'info, SplAccountCompres
    pub bubblegum_program: Program<'info, MplBubblegum>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct WithdrawTwo<'info> {
    #[account(
        seeds = [merkle_tree1.key().as_ref()],
        bump,
        seeds::program = bubblegum_program.key()
    )]
    /// CHECK: This account is neither written to nor read fr
    pub tree_authority1: Account<'info, TreeConfig>,
    #[account(
        seeds = [b"cNFT-vault"],
        bump,
    )]
    /// CHECK: This account doesnt even exist (it is just the
    pub leaf_owner: UncheckedAccount<'info>, // you might nee
    /// CHECK: This account is neither written to nor read fr
    pub new_leaf_owner1: UncheckedAccount<'info>, // receiver
    #[account(mut)]
    /// CHECK: This account is modified in the downstream pro
    pub merkle_tree1: UncheckedAccount<'info>,

    #[account(
        seeds = [merkle_tree2.key().as_ref()],
        bump,
        seeds::program = bubblegum_program.key()
    )]
    /// CHECK: This account is neither written to nor read fr
    pub tree_authority2: Account<'info, TreeConfig>,
    /// CHECK: This account is neither written to nor read fr
    pub new_leaf_owner2: UncheckedAccount<'info>, // receiver
    #[account(mut)]
    /// CHECK: This account is modified in the downstream pro
    pub merkle_tree2: UncheckedAccount<'info>,

    pub log_wrapper: Program<'info, Noop>,
    pub compression_program: Program<'info, SplAccountCompres
    pub bubblegum_program: Program<'info, MplBubblegum>,
    pub system_program: Program<'info, System>,
}
```

program_actions.rs

Solana program with access control logic, defining actions like 'mint' and 'verify', and integration with Metaplex Bubblegum.

```
#![allow(clippy::result_large_err)]

pub mod actions;
pub use actions::*;

pub mod state;
pub use state::*;

use anchor_lang::prelude::*;
use solana_program::pubkey::Pubkey;
use spl_account_compression::{program::SplAccountCompression,

#[derive(Clone)]
pub struct MplBubblegum;

impl anchor_lang::Id for MplBubblegum {
    fn id() -> Pubkey {
        mpl_bubblegum::id()
    }
}

declare_id!("burZc1SfqbrAP35XG63YZZ82C9Zd22QUwhCXoEUZWNF");

#[program]
pub mod cutils {
    use super::*;

    #[access_control(ctx.accounts.validate(&ctx, &params))]
    pub fn mint<'info>(
        ctx: Context<'_, '_, '_, 'info, Mint<'info>>,
        params: MintParams,
    ) -> Result<()> {
        Mint::actuate(ctx, params)
    }

    #[access_control(ctx.accounts.validate(&ctx, &params))]
    pub fn verify<'info>(
        ctx: Context<'_, '_, '_, 'info, Verify<'info>>,
        params: VerifyParams,
    ) -> Result<()> {
        Verify::actuate(ctx, &params)
    }
}
```

pyth_price_reader.rs

Solana program retrieving and processing price data from the Pyth oracle network, including error handling.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
pub mod state;
use state::PriceFeed;

pub mod error;
```

```
use error::ErrorCode;

declare_id!("F6mNuN1xoPdRaZcUX3Xviq7x1EFtoBXygpFggCLd62eU");

#[program]
pub mod pythexample {
    use super::*;
    pub fn read_price(ctx: Context<Pyth>) -> Result<()> {
        let price_feed = &ctx.accounts.price_feed;
        let clock = &ctx.accounts.clock;
        // Get the current timestamp
        let timestamp: i64 = clock.unix_timestamp;
        // Load the price from the price feed. Here, the pric
        let price: pyth_sdk::Price = price_feed
            .get_price_no_older_than(timestamp, 30)
            .ok_or(ErrorCode::PythError)?;

        let confidence_interval: u64 = price.conf;

        let asset_price_full: i64 = price.price;

        let asset_exponent: i32 = price.expo;

        let asset_price = asset_price_full as f64 * 10f64.pow

        msg!("Price: {}", asset_price);
        msg!("Confidence interval: {}", confidence_interval);

        Ok(())
    }
}

#[derive(Accounts)]
pub struct Pyth<'info> {
    pub price_feed: Account<'info, PriceFeed>,
    pub system_program: Program<'info, System>,
    pub clock: Sysvar<'info, Clock>,
}
```

change_package_version.ts

Node.js function to modify the version of a specified package within `package.json`'s "dependencies" or "devDependencies".

```
import { readFileSync } from 'node:fs'

export function changePackageVersion(file: string, pkgName: s
  const content = JSON.parse(readFileSync(file).toString('utf
  if (content.dependencies && content.dependencies[pkgName] &
    content.dependencies[pkgName] = pkgVersion
    return [true, content]
  }
  if (content.devDependencies && content.devDependencies[pkgN
    content.devDependencies[pkgName] = pkgVersion
    return [true, content]
  }
  return [false, content]
}
```

dependency_analyzer.ts

Node.js script that scans a project directory for `package.json` files, identifies duplicate dependencies with multiple versions, and reports them.

```
import { basename } from 'node:path'
import * as p from 'picocolors'
import { getDepsCount } from './get-deps-count'
import { getRecursiveFileList } from './get-recursive-file-li

export function commandCheck(path: string = '.') {
  const files = getRecursiveFileList(path).filter((file) => b
  const depsCounter = getDepsCount(files)

  const single: string[] = []
  const multiple: string[] = []

  Object.keys(depsCounter)
    .sort()
    .map((pkg) => {
      const versions = depsCounter[pkg]
      const versionMap = Object.keys(versions).sort()
      const versionsLength = versionMap.length

      if (versionsLength === 1) {
        const count = versions[versionMap[0]].length
        single.push(`${p.green(`✔`)} ${pkg}@${versionMap[0]}
        return
      }

      const versionCount: { version: string; count: number }[
      for (const version of versionMap) {
        versionCount.push({ version, count: versions[version]
      }
      versionCount.sort((a, b) => b.count - a.count)

      multiple.push(`${p.yellow(`⚠`)} ${pkg} has ${versionsLe

      for (const { count, version } of versionCount) {
        multiple.push(`  - ${p.bold(version)} (${count})`)
      }
    })

  for (const string of [...single.sort(), ...multiple]) {
    console.log(string)
  }
}
```

sync-package-json.ts

Displays a help menu for a 'sync-package-json' command-line tool, including commands, options, and examples. `Usage: yarn sync-package-json <command> [options]`

```
export function commandHelp() {
  console.log(`Usage: yarn sync-package-json <command> [optio
  console.log(``)
  console.log(`Commands:`)
  console.log(`  check  <path>        Check package.json file
  console.log(`  help                 Show this help`)
  console.log(`  list   <path>        List package.json files
  console.log(`  set    [ver] <path>  Set specific version in
  console.log(`  update <path> <pkgs> Update all versions in
  console.log(``)
  console.log(`Arguments:`)
  console.log(`  path    Path to directory`)
  console.log(``)
  console.log(`Examples:`)
  console.log(`  yarn sync-package-json check`)
  console.log(`  yarn sync-package-json check basics`)
  console.log(`  yarn sync-package-json list`)
  console.log(`  yarn sync-package-json list basics`)
  console.log(`  yarn sync-package-json help`)
  console.log(`  yarn sync-package-json set @coral-xyz/anchor
  console.log(`  yarn sync-package-json set @coral-xyz/anchor
  console.log(`  yarn sync-package-json update`)
  console.log(`  yarn sync-package-json update basics`)
  console.log(`  yarn sync-package-json update . @solana/web3
  process.exit(0)
}
```

get_command_list.ts

Node.js function that lists the paths to all `package.json` files found recursively within a specified directory.

```
import { basename } from 'node:path'
import { getRecursiveFileList } from './get-recursive-file-li

export function commandList(path: string) {
  const files = getRecursiveFileList(path).filter((file) => b
  for (const file of files) {
    console.log(file)
  }
}
```

command-set.ts

Node.js function designed to update the specified version of a package within all `package.json` files found in a directory tree.

```
import { writeFileSync } from 'fs'
import { basename } from 'node:path'
import { changePackageVersion } from './change-package-versio
import { getRecursiveFileList } from './get-recursive-file-li

export function commandSet(version: string, path: string = '.
  if (!version) {
    console.error(`Version is required`)
    process.exit(1)
  }
  if (
    !version
      // Strip first character if it's a `@`
      .replace(/^@/, '')
      .includes('@')
  ) {
    console.error(`Invalid package version: ${version}. Provi
    process.exit(1)
  }
  // Take anything after the second `@` as the version, the r
  const [pkg, ...rest] = version.split('@').reverse()
  const pkgName = rest.reverse().join('@')

  // Make sure pkgVersions has a ^ prefix, if not add it
  const pkgVersion = pkg.startsWith('^') ? pkg : `^${pkg}`

  console.log(`Setting package ${pkgName} to ${pkgVersion} in

  const files = getRecursiveFileList(path).filter((file) => b
  let count = 0
  for (const file of files) {
    const [changed, content] = changePackageVersion(file, pkg
    if (changed) {
      writeFileSync(file, JSON.stringify(content, null, 2) +
      count++
    }
  }
  if (count === 0) {
    console.log(`No files updated`)
  } else {
    console.log(`Updated ${count} files`)
  }
}
```

command_update.ts

Node.js function designed to update packages in `package.json` files within a directory, either all packages or a specified list, to their latest versions.

```
import { execSync } from 'child_process'
import { writeFileSync } from 'fs'
import { basename } from 'node:path'
import * as p from 'picocolors'
import { changePackageVersion } from './change-package-versio

import { getDepsCount } from './get-deps-count'
import { getRecursiveFileList } from './get-recursive-file-li

export function commandUpdate(path: string = '.', packageName
  const files = getRecursiveFileList(path).filter((file) => b
  const depsCounter = getDepsCount(files)
  const pkgNames = Object.keys(depsCounter).sort()
  if (packageNames.length > 0) {
    console.log(`Updating ${packageNames.join(', ')} in ${fil
  }

  let total = 0
  for (const pkgName of pkgNames.filter((pkgName) => packageN
    // Get latest version from npm
    const npmVersion = execSync(`npm view ${pkgName} version`

    let count = 0
    for (const file of files) {
      const [changed, content] = changePackageVersion(file, p
      if (changed) {
        writeFileSync(file, JSON.stringify(content, null, 2) 
        count++
      }
    }
    total += count

    if (count === 0) {
      console.log(p.dim(`Package ${pkgName} is up to date ${n
      continue
    }
    console.log(p.green(` -> Updated ${count} files with ${pk
  }

  if (total === 0) {
    console.log(`No files updated`)
  } else {
    console.log(`Updated ${total} files`)
  }
}
```

get_dependency_count.ts

Node.js function that analyzes a list of `package.json` files, builds a map of dependencies, versions, and the files where they are found.

```
import { readFileSync } from 'node:fs'

export function getDepsCount(files: string[] = []): Record<st
  const map: Record<string, JSON> = {}
  const depsCounter: Record<string, Record<string, string[]>>

  for (const file of files) {
    const content = JSON.parse(readFileSync(file).toString('u
    map[file] = content

    const deps = content.dependencies ?? {}
    const devDeps = content.devDependencies ?? {}
    const merged = { ...deps, ...devDeps }

    Object.keys(merged)
      .sort()
      .map((pkg) => {
        const pkgVersion = merged[pkg]
        if (!depsCounter[pkg]) {
          depsCounter[pkg] = { [pkgVersion]: [file] }
          return
        }
        if (!depsCounter[pkg][pkgVersion]) {
          depsCounter[pkg][pkgVersion] = [file]
          return
        }
        depsCounter[pkg][pkgVersion] = [...depsCounter[pkg][p
      })
  }
  return depsCounter
}
```

get_recursive_file_list.ts

Node.js function to recursively retrieve a list of file paths within a directory, excluding common development-related folders.

```
// Point method at path and return a list of all the files in
import { readdirSync, statSync } from 'node:fs'

export function getRecursiveFileList(path: string): string[]
  const ignore = ['.git', '.github', '.idea', '.next', '.verc
  const files: string[] = []

  const items = readdirSync(path)
  items.forEach((item) => {
    if (ignore.includes(item)) {
      return
    }
    // Check out if it's a directory or a file
    const isDir = statSync(`${path}/${item}`).isDirectory()
    if (isDir) {
      // If it's a directory, recursively call the method
      files.push(...getRecursiveFileList(`${path}/${item}`))
    } else {
      // If it's a file, add it to the array of files
      files.push(`${path}/${item}`)
    }
  })

  return files.filter((file) => {
    // Remove package.json from the root directory
    return path === '.' ? file !== './package.json' : true
  })
}
```

createToken.rs

Solana program enabling users to create a new token mint along with its corresponding metadata.

```
#![allow(clippy::result_large_err)]

use {
    anchor_lang::prelude::*,
    anchor_spl::{
        metadata::{create_metadata_accounts_v3, CreateMetadat
        token::{Mint, Token},
    },
    mpl_token_metadata::{pda::find_metadata_account, state::D
};

declare_id!("2B6MrsKB2pVq6W6tY8dJLcnSd3Uv1KE7yRaboBjdQoEX");

#[program]
pub mod create_token {
    use super::*;

    pub fn create_token_mint(
        ctx: Context<CreateTokenMint>,
        token_name: String,
        token_symbol: String,
        token_uri: String,
        _token_decimals: u8,
    ) -> Result<()> {
        msg!("Creating metadata account...");
        msg!(
            "Metadata account address: {}",
            &ctx.accounts.metadata_account.key()
        );

        // Cross Program Invocation (CPI)
        // Invoking the create_metadata_account_v3 instructio
        create_metadata_accounts_v3(
            CpiContext::new(
                ctx.accounts.token_metadata_program.to_accoun
                CreateMetadataAccountsV3 {
                    metadata: ctx.accounts.metadata_account.t
                    mint: ctx.accounts.mint_account.to_accoun
                    mint_authority: ctx.accounts.payer.to_acc
                    update_authority: ctx.accounts.payer.to_a
                    payer: ctx.accounts.payer.to_account_info
                    system_program: ctx.accounts.system_progr
                    rent: ctx.accounts.rent.to_account_info()
                },
            ),
            DataV2 {
                name: token_name,
                symbol: token_symbol,
                uri: token_uri,
                seller_fee_basis_points: 0,
                creators: None,
                collection: None,
                uses: None,
            },
            false, // Is mutable
            true,  // Update authority is signer
            None,  // Collection details
        )?;

        msg!("Token mint created successfully.");

        Ok(())
    }
}

#[derive(Accounts)]
#[instruction(_token_decimals: u8)]
pub struct CreateTokenMint<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    /// CHECK: Address validated using constraint
    #[account(
        mut,
        address=find_metadata_account(&mint_account.key()).0
    )]
    pub metadata_account: UncheckedAccount<'info>,
    // Create new mint account
    #[account(
        init,
        payer = payer,
        mint::decimals = _token_decimals,
        mint::authority = payer.key(),
    )]
    pub mint_account: Account<'info, Mint>,

    pub token_metadata_program: Program<'info, Metadata>,
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}
```

nft_minting.rs

Solana program enabling users to mint an NFT, including token creation, metadata setup, and master edition generation.

```
#![allow(clippy::result_large_err)]

use {
    anchor_lang::prelude::*,
    anchor_spl::{
        associated_token::AssociatedToken,
        metadata::{
            create_master_edition_v3, create_metadata_account
            CreateMetadataAccountsV3, Metadata,
        },
        token::{mint_to, Mint, MintTo, Token, TokenAccount},
    },
    mpl_token_metadata::{
        pda::{find_master_edition_account, find_metadata_acco
        state::DataV2,
    },
};

declare_id!("3qHNM98iLTaQtwmj2NkViXnHZQjNBS5PTHT2AuPxHXYN");

#[program]
pub mod nft_minter {
    use super::*;

    pub fn mint_nft(
        ctx: Context<CreateToken>,
        nft_name: String,
        nft_symbol: String,
        nft_uri: String,
    ) -> Result<()> {
        msg!("Minting Token");
        // Cross Program Invocation (CPI)
        // Invoking the mint_to instruction on the token prog
        mint_to(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                MintTo {
                    mint: ctx.accounts.mint_account.to_accoun
                    to: ctx.accounts.associated_token_account
                    authority: ctx.accounts.payer.to_account_
                },
            ),
            1,
        )?;

        msg!("Creating metadata account");
        // Cross Program Invocation (CPI)
        // Invoking the create_metadata_account_v3 instructio
        create_metadata_accounts_v3(
            CpiContext::new(
                ctx.accounts.token_metadata_program.to_accoun
                CreateMetadataAccountsV3 {
                    metadata: ctx.accounts.metadata_account.t
                    mint: ctx.accounts.mint_account.to_accoun
                    mint_authority: ctx.accounts.payer.to_acc
                    update_authority: ctx.accounts.payer.to_a
                    payer: ctx.accounts.payer.to_account_info
                    system_program: ctx.accounts.system_progr
                    rent: ctx.accounts.rent.to_account_info()
                },
            ),
            DataV2 {
                name: nft_name,
                symbol: nft_symbol,
                uri: nft_uri,
                seller_fee_basis_points: 0,
                creators: None,
                collection: None,
                uses: None,
            },
            false, // Is mutable
            true,  // Update authority is signer
            None,  // Collection details
        )?;

        msg!("Creating master edition account");
        // Cross Program Invocation (CPI)
        // Invoking the create_master_edition_v3 instruction o
        create_master_edition_v3(
            CpiContext::new(
                ctx.accounts.token_metadata_program.to_accoun
                CreateMasterEditionV3 {
                    edition: ctx.accounts.edition_account.to_a
                    mint: ctx.accounts.mint_account.to_accoun
                    update_authority: ctx.accounts.payer.to_ac
                    mint_authority: ctx.accounts.payer.to_acco
                    payer: ctx.accounts.payer.to_account_info
                    metadata: ctx.accounts.metadata_account.to
                    token_program: ctx.accounts.token_program
                    system_program: ctx.accounts.system_progra
                    rent: ctx.accounts.rent.to_account_info()
                },
            ),
            None, // Max Supply
        )?;

        msg!("NFT minted successfully.");

        Ok(())
    }
}

#[derive(Accounts)]
pub struct CreateToken<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    /// CHECK: Address validated using constraint
    #[account(
        mut,
        address=find_metadata_account(&mint_account.key()).0
    )]
    pub metadata_account: UncheckedAccount<'info>,

    /// CHECK: Address validated using constraint
    #[account(
        mut,
        address=find_master_edition_account(&mint_account.key
    )]
    pub edition_account: UncheckedAccount<'info>,

    // Create new mint account, NFTs have 0 decimals
    #[account(
        init,
        payer = payer,
        mint::decimals = 0,
        mint::authority = payer.key(),
        mint::freeze_authority = payer.key(),
    )]
    pub mint_account: Account<'info, Mint>,

    // Create associated token account, if needed
    // This is the account that will hold the NFT
    #[account(
        init_if_needed,
        payer = payer,
        associated_token::mint = mint_account,
        associated_token::authority = payer,
    )]
    pub associated_token_account: Account<'info, TokenAccount

    pub token_program: Program<'info, Token>,
    pub token_metadata_program: Program<'info, Metadata>,
    pub associated_token_program: Program<'info, AssociatedTo
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}
```

pda-mint-authority.rs

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use instructions::*;
pub mod instructions;

declare_id!("A5gNtapBvMLD6i7D2t3SSyJeFtBdfb6ibvZu1hoBLzCo");

#[program]
pub mod token_minter {
    use super::*;

    pub fn create_token(
        ctx: Context<CreateToken>,
        token_name: String,
        token_symbol: String,
        token_uri: String,
    ) -> Result<()> {
        create::create_token(ctx, token_name, token_symbol, t
    }

    pub fn mint_token(ctx: Context<MintToken>, amount: u64) -
        mint::mint_token(ctx, amount)
    }
}
```

spl-token-minter.rs

Solana program for creating and minting SPL (Solana Program Library) tokens. It includes two instructions: `create_token` to create a new SPL token with a name, symbol, and URI, and `mint_token` to mint a specified amount of an existing SPL token.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

pub mod instructions;
```

```rust
use instructions::*;

declare_id!("77p9WmpzQW29RUEzTEef2ym7AHePBE9yNWJ9acikXfZS");

#[program]
pub mod spl_token_minter {
    use super::*;

    pub fn create_token(
        ctx: Context<CreateToken>,
        token_name: String,
        token_symbol: String,
        token_uri: String,
    ) -> Result<()> {
        create::create_token(ctx, token_name, token_symbol, t
    }

    pub fn mint_token(ctx: Context<MintToken>, amount: u64) -
        mint::mint_token(ctx, amount)
    }
}
```

anchor-funcs.rs

Solana program built with the Anchor framework for creating and interacting with SPL (Solana Program Library) tokens. It includes instructions for creating a token, creating a token account, creating an associated token account, transferring tokens, and minting tokens. The program utilizes Anchor's account derivation and CPI (Cross-Program Invocation) to interact with the Solana Token Program.

```rust
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;
use anchor_spl::associated_token::AssociatedToken;
use anchor_spl::token_interface::{
    self, Mint, MintTo, TokenAccount, TokenInterface, Transfe
};
```

```
declare_id!("6qNqxkRF791FXFeQwqYQLEzAbGiqDULC5SSHVsfRoG89");

#[program]
pub mod anchor {

    use super::*;

    pub fn create_token(_ctx: Context<CreateToken>, _token_na
        msg!("Create Token");
        Ok(())
    }
    pub fn create_token_account(_ctx: Context<CreateTokenAcco
        msg!("Create Token Account");
        Ok(())
    }
    pub fn create_associated_token_account(
        _ctx: Context<CreateAssociatedTokenAccount>,
    ) -> Result<()> {
        msg!("Create Associated Token Account");
        Ok(())
    }
    pub fn transfer_token(ctx: Context<TransferToken>, amount
        let cpi_accounts = TransferChecked {
            from: ctx.accounts.from.to_account_info().clone()
            mint: ctx.accounts.mint.to_account_info().clone()
            to: ctx.accounts.to_ata.to_account_info().clone()
            authority: ctx.accounts.signer.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_accou
        let cpi_context = CpiContext::new(cpi_program, cpi_ac
        token_interface::transfer_checked(cpi_context, amount
        msg!("Transfer Token");
        Ok(())
    }
    pub fn mint_token(ctx: Context<MintToken>, amount: u64) -
        let cpi_accounts = MintTo {
            mint: ctx.accounts.mint.to_account_info().clone()
            to: ctx.accounts.receiver.to_account_info().clone
            authority: ctx.accounts.signer.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_accou
        let cpi_context = CpiContext::new(cpi_program, cpi_ac
        token_interface::mint_to(cpi_context, amount)?;
        msg!("Mint Token");
        Ok(())
    }
}

#[derive(Accounts)]
#[instruction(token_name: String)]
pub struct CreateToken<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    #[account(
        init,
        payer = signer,
        mint::decimals = 6,
        mint::authority = signer.key(),
        seeds = [b"token-2022-token", signer.key().as_ref(),
        bump,
    )]
    pub mint: InterfaceAccount<'info, Mint>,
    pub system_program: Program<'info, System>,
    pub token_program: Interface<'info, TokenInterface>,
}

#[derive(Accounts)]
pub struct CreateTokenAccount<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        init,
        token::mint = mint,
        token::authority = signer,
        payer = signer,
        seeds = [b"token-2022-token-account", signer.key().as_
        bump,
    )]
    pub token_account: InterfaceAccount<'info, TokenAccount>,
    pub system_program: Program<'info, System>,
    pub token_program: Interface<'info, TokenInterface>,
}

#[derive(Accounts)]
pub struct CreateAssociatedTokenAccount<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        init,
        associated_token::mint = mint,
        payer = signer,
        associated_token::authority = signer,
    )]
    pub token_account: InterfaceAccount<'info, TokenAccount>,
    pub system_program: Program<'info, System>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
}

#[derive(Accounts)]

pub struct TransferToken<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    #[account(mut)]
    pub from: InterfaceAccount<'info, TokenAccount>,
    pub to: SystemAccount<'info>,
    #[account(
        init,
        associated_token::mint = mint,
        payer = signer,
        associated_token::authority = to
    )]
    pub to_ata: InterfaceAccount<'info, TokenAccount>,
    #[account(mut)]
    pub mint: InterfaceAccount<'info, Mint>,
    pub token_program: Interface<'info, TokenInterface>,
    pub system_program: Program<'info, System>,
    pub associated_token_program: Program<'info, AssociatedTo
}

#[derive(Accounts)]
pub struct MintToken<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    #[account(mut)]
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(mut)]
    pub receiver: InterfaceAccount<'info, TokenAccount>,
    pub token_program: Interface<'info, TokenInterface>,
}
```

spl-process.rs

Solana program that creates a new SPL token mint account with a specified number of decimals by interacting with the Token Program through cross-program invocations.

```
use {
    borsh::{BorshDeserialize, BorshSerialize},
    solana_program::{
        account_info::{next_account_info, AccountInfo},
        entrypoint,
        entrypoint::ProgramResult,
        msg,
        program::invoke,
        pubkey::Pubkey,
        rent::Rent,
        system_instruction,
        sysvar::Sysvar,
    },
    spl_token_2022::{
        extension::{
            default_account_state::instruction::{
                initialize_default_account_state, update_defa
            },
            ExtensionType,
        },
        instruction as token_instruction,
        state::AccountState,
        state::Mint,
    },
};

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CreateTokenArgs {
    pub token_decimals: u8,
}

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let args = CreateTokenArgs::try_from_slice(instruction_da

    let accounts_iter = &mut accounts.iter();

    let mint_account = next_account_info(accounts_iter)?;
    let mint_authority = next_account_info(accounts_iter)?;
    let payer = next_account_info(accounts_iter)?;
    let rent = next_account_info(accounts_iter)?;
    let system_program = next_account_info(accounts_iter)?;
    let token_program = next_account_info(accounts_iter)?;

    // Find the size for the account with the Extension
    let space = ExtensionType::get_account_len::<Mint>(&[Exte
    // Get the required rent exemption amount for the account
    let rent_required = Rent::get()?.minimum_balance(space);

    // Create the account for the Mint and allocate space

    msg!("Mint account address : {}", mint_account.key);
    invoke(
        &system_instruction::create_account(
            payer.key,
            mint_account.key,
            rent_required,
            space as u64,
            token_program.key,
        ),
        &[
            mint_account.clone(),
            payer.clone(),
            system_program.clone(),
            token_program.clone(),
        ],
    )?;

    // This needs to be done before the Mint is initialized

    // Initialize the Default Account State as Frozen
    invoke(
        &initialize_default_account_state(
            token_program.key,
            mint_account.key,
            &AccountState::Frozen,
        )
        .unwrap(),
        &[
            mint_account.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Token Mint
    invoke(
        &token_instruction::initialize_mint(
            token_program.key,
            mint_account.key,
            mint_authority.key,
            Some(mint_authority.key),
            args.token_decimals,
        )?,
        &[
            mint_account.clone(),
            mint_authority.clone(),
            token_program.clone(),
            rent.clone(),
        ],
    )?;

    // Update the Default Account State to Initialized
    invoke(
        &update_default_account_state(
            token_program.key,
            mint_account.key,
            payer.key,
            &[payer.key],
            &AccountState::Initialized,
        )
        .unwrap(),
        &[
            mint_account.clone(),
            payer.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    msg!("Mint created!");
    Ok(())
}
```

mint-close-authority.rs

Solana program to create a new SPL token mint account with a specified number of decimals, initializes the mint close authority extension, and sets the mint authority. It interacts with the Token Program and System Program through cross-program invocations.

```
use {
    borsh::{BorshDeserialize, BorshSerialize},
    solana_program::{
        account_info::{next_account_info, AccountInfo},
        entrypoint,
        entrypoint::ProgramResult,
        msg,
        program::invoke,
        pubkey::Pubkey,
        rent::Rent,
        system_instruction,
        sysvar::Sysvar,
    },
    spl_token_2022::{extension::ExtensionType, instruction as
};

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CreateTokenArgs {
    pub token_decimals: u8,
}

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let args = CreateTokenArgs::try_from_slice(instruction_da

    let accounts_iter = &mut accounts.iter();

    let mint_account = next_account_info(accounts_iter)?;
    let mint_authority = next_account_info(accounts_iter)?;
    let close_authority = next_account_info(accounts_iter)?;
    let payer = next_account_info(accounts_iter)?;
    let rent = next_account_info(accounts_iter)?;
    let system_program = next_account_info(accounts_iter)?;
    let token_program = next_account_info(accounts_iter)?;

    // Find the size for the account with the Extension
    let space = ExtensionType::get_account_len::<Mint>(&[Exte

    // Get the required rent exemption amount for the account
    let rent_required = Rent::get()?.minimum_balance(space);

    // Create the account for the Mint and allocate space
    msg!("Mint account address : {}", mint_account.key);
    invoke(
        &system_instruction::create_account(
            payer.key,
            mint_account.key,
            rent_required,
            space as u64,
            token_program.key,
        ),
        &[
            mint_account.clone(),
            payer.clone(),
            system_program.clone(),
            token_program.clone(),
        ],
    )?;

    // This needs to be done before the Mint is initialized
    // Initialize the Mint close authority Extension
    invoke(
        &token_instruction::initialize_mint_close_authority(
            token_program.key,
            mint_account.key,
            Some(close_authority.key),
        )
        .unwrap(),
        &[
            mint_account.clone(),
            close_authority.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Token Mint
    invoke(
        &token_instruction::initialize_mint(
            token_program.key,
            mint_account.key,
            mint_authority.key,
            Some(mint_authority.key),
            args.token_decimals,
        )?,
        &[
            mint_account.clone(),
            mint_authority.clone(),
            token_program.clone(),
            rent.clone(),
        ],
    )?;

    msg!("Mint created!");

    Ok(())
}
```

multiple-extension-mint-authority.rs

Solana program to create a new SPL token mint account with a specified number of decimals, multiple extensions, initializes the mint close authority and non-transferable mint extensions, and sets the mint authority. It interacts with the Token Program and System Program through cross-program invocations to enable advanced features on the mint.

```
use {
    borsh::{BorshDeserialize, BorshSerialize},
    solana_program::{
        account_info::{next_account_info, AccountInfo},
        entrypoint,
        entrypoint::ProgramResult,
        msg,
        program::invoke,
        pubkey::Pubkey,
        rent::Rent,
        system_instruction,
        sysvar::Sysvar,
    },
    spl_token_2022::{extension::ExtensionType, instruction as
};

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CreateTokenArgs {
    pub token_decimals: u8,
}

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let args = CreateTokenArgs::try_from_slice(instruction_da

    let accounts_iter = &mut accounts.iter();
```

```rust
let mint_account = next_account_info(accounts_iter)?;
let mint_authority = next_account_info(accounts_iter)?;
let close_authority = next_account_info(accounts_iter)?;
let payer = next_account_info(accounts_iter)?;
let rent = next_account_info(accounts_iter)?;
let system_program = next_account_info(accounts_iter)?;
let token_program = next_account_info(accounts_iter)?;

// Find the size for the Mint account with the the number
let space = ExtensionType::get_account_len::<Mint>(&[
    ExtensionType::MintCloseAuthority,
    ExtensionType::NonTransferable,
]);

// Get the required rent exemption amount for the account
let rent_required = Rent::get()?.minimum_balance(space);

// Create the account for the Mint and allocate space
msg!("Mint account address : {}", mint_account.key);
invoke(
    &system_instruction::create_account(
        payer.key,
        mint_account.key,
        rent_required,
        space as u64,
        token_program.key,
    ),
    &[
        mint_account.clone(),
        payer.clone(),
        system_program.clone(),
        token_program.clone(),
    ],
)?;

// Here, let's enable two extensions for the Mint. This n
```

```
    // Initialize the Mint close authority Extension
    invoke(
        &token_instruction::initialize_mint_close_authority(
            token_program.key,
            mint_account.key,
            Some(close_authority.key),
        )
        .unwrap(),
        &[
            mint_account.clone(),
            close_authority.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Non Transferable Mint Extension
    invoke(
        &token_instruction::initialize_non_transferable_mint(
            .unwrap(),
        &[
            mint_account.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Token Mint
    invoke(
        &token_instruction::initialize_mint(
            token_program.key,
            mint_account.key,
            mint_authority.key,
            Some(mint_authority.key),
            args.token_decimals,
        )?,
        &[
            mint_account.clone(),
            mint_authority.clone(),
            token_program.clone(),
            rent.clone(),
        ],
    )?;

    msg!("Mint created!");

    Ok(())
}
```

nft-metadata-pointer-lib.rs

This is the main entry point for a Solana program built with Anchor. It defines instructions for initializing players, performing game actions like chopping trees, and minting NFTs, with session-based authentication. Note that the instructions, error handling, state files are written separately.

```
pub use crate::errors::GameErrorCode;
pub use anchor_lang::prelude::*;
pub use session_keys::{session_auth_or, Session, SessionError
pub mod constants;
pub mod errors;
pub mod instructions;
pub mod state;
use instructions::*;

declare_id!("H31ofLpWqeAzF2Pg54HSPQGYifJad843tTJg8vCYVoh3");

#[program]
pub mod extension_nft {

    use super::*;

    pub fn init_player(ctx: Context<InitPlayer>, _level_seed:
        init_player::init_player(ctx)
    }

    // This function lets the player chop a tree and get 1 wo
    // lets the player either use their session token or thei
    // there so that the player can do multiple transactions 
    // in the same block would result in the same signature a
    #[session_auth_or(
        ctx.accounts.player.authority.key() == ctx.accounts.s
        GameErrorCode::WrongAuthority
    )]
    pub fn chop_tree(ctx: Context<ChopTree>, _level_seed: Str
        chop_tree::chop_tree(ctx, counter, 1)
    }

    pub fn mint_nft(ctx: Context<MintNft>) -> Result<()> {
        mint_nft::mint_nft(ctx)
    }
}
```

non-transferable.rs

Solana program to create a new non-transferable SPL token mint account with a specified number of decimals. It initializes the non-transferable mint extension before creating the mint account, ensuring that the minted tokens cannot be transferred.

```
use {
    borsh::{BorshDeserialize, BorshSerialize},
    solana_program::{
        account_info::{next_account_info, AccountInfo},
        entrypoint,
        entrypoint::ProgramResult,
        msg,
        program::invoke,
        pubkey::Pubkey,
        rent::Rent,
        system_instruction,
        sysvar::Sysvar,
    },
    spl_token_2022::{extension::ExtensionType, instruction as
};
```

```
#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CreateTokenArgs {
    pub token_decimals: u8,
}

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let args = CreateTokenArgs::try_from_slice(instruction_da

    let accounts_iter = &mut accounts.iter();

    let mint_account = next_account_info(accounts_iter)?;
    let mint_authority = next_account_info(accounts_iter)?;
    let payer = next_account_info(accounts_iter)?;
    let rent = next_account_info(accounts_iter)?;
    let system_program = next_account_info(accounts_iter)?;
    let token_program = next_account_info(accounts_iter)?;

    // Find the size for the account with the Extension
    let space = ExtensionType::get_account_len::<Mint>(&[Exte

    // Get the required rent exemption amount for the account
    let rent_required = Rent::get()?.minimum_balance(space);

    // Create the account for the Mint and allocate space
    msg!("Mint account address : {}", mint_account.key);
    invoke(
        &system_instruction::create_account(
            payer.key,
            mint_account.key,
            rent_required,
            space as u64,
            token_program.key,
        ),
        &[
            mint_account.clone(),
            payer.clone(),
            system_program.clone(),
            token_program.clone(),
        ],
    )?;

    // This needs to be done before the Mint is initialized

    // Initialize the Non Transferable Mint Extension
    invoke(
        &token_instruction::initialize_non_transferable_mint(
            .unwrap(),
        &[
            mint_account.clone(),
            token_program.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Token Mint
    invoke(
        &token_instruction::initialize_mint(
            token_program.key,
            mint_account.key,
            mint_authority.key,
            Some(mint_authority.key),
            args.token_decimals,
        )?,
        &[
            mint_account.clone(),
            mint_authority.clone(),
            token_program.clone(),
            rent.clone(),
        ],
    )?;
    msg!("Mint created!");

    Ok(())
}
```

transfer-fee.rs

Solana program to create a new SPL token mint account with a specified number of decimals and initializes the transfer fee extension. It sets up a transfer fee configuration where a fixed percentage (10% in the example) of tokens will be charged as a fee for every token transfer. The maximum fee amount is also set.

```
use {
    borsh::{BorshDeserialize, BorshSerialize},
    solana_program::{
        account_info::{next_account_info, AccountInfo},
        entrypoint,
        entrypoint::ProgramResult,
        msg,
        program::invoke,
        pubkey::Pubkey,
        rent::Rent,
        system_instruction,
        sysvar::Sysvar,
    },
    spl_token_2022::{
        extension::{
            transfer_fee::instruction::{initialize_transfer_f
            ExtensionType,
        },
        instruction as token_instruction,
        state::Mint,
    },
};

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CreateTokenArgs {
    pub token_decimals: u8,
}

entrypoint!(process_instruction);

fn process_instruction(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let args = CreateTokenArgs::try_from_slice(instruction_da

    let accounts_iter = &mut accounts.iter();

    let mint_account = next_account_info(accounts_iter)?;
    let mint_authority = next_account_info(accounts_iter)?;
    let payer = next_account_info(accounts_iter)?;
    let rent = next_account_info(accounts_iter)?;
    let system_program = next_account_info(accounts_iter)?;
    let token_program = next_account_info(accounts_iter)?;

    // Find the size for the account with the Extension
    let space = ExtensionType::get_account_len::<Mint>(&[Exte

    // Get the required rent exemption amount for the account
    let rent_required = Rent::get()?.minimum_balance(space);

    // Create the account for the Mint and allocate space
    msg!("Mint account address : {}", mint_account.key);
    invoke(
        &system_instruction::create_account(
            payer.key,
            mint_account.key,
            rent_required,
            space as u64,
            token_program.key,
        ),
        &[
            mint_account.clone(),
            payer.clone(),
            system_program.clone(),
            token_program.clone(),
        ],
    )?;

    // The max fee will be 5 tokens, here we adjust it with t
    let max_fee = 5 * 10u64.pow(args.token_decimals as u32);

    // This needs to be done before the Mint is initialized
    // Initialize the Transfer Fee config
    invoke(
        &initialize_transfer_fee_config(
            token_program.key,
            mint_account.key,
            Some(payer.key),
            Some(payer.key),
            // 1% fee on transfers
            100,
            max_fee,
        )
        .unwrap(),
        &[
            mint_account.clone(),
            token_program.clone(),
            payer.clone(),
            system_program.clone(),
        ],
    )?;

    // Initialize the Token Mint
    invoke(
        &token_instruction::initialize_mint(
            token_program.key,
            mint_account.key,
            mint_authority.key,
            Some(mint_authority.key),
            args.token_decimals,
        )?,
        &[
            mint_account.clone(),
            mint_authority.clone(),
            token_program.clone(),
            rent.clone(),
        ],
    )?;

    // Initialize the Transfer Fee config
    invoke(
        &set_transfer_fee(
            token_program.key,
            mint_account.key,
            payer.key,
            &[payer.key],
            // 10% fee on transfers
            1000,
            max_fee,
        )
        .unwrap(),
        &[
            mint_account.clone(),
            token_program.clone(),
            payer.clone(),
            system_program.clone(),
        ],
    )?;

    msg!("Mint created!");

    Ok(())
}
```

transferhook.rs

This Solana program built with the Anchor framework defines instructions for initializing an extra account metadata list and implementing a custom transfer hook. The transfer hook instruction allows tracking the number of times a token has been transferred by incrementing a counter account. It utilizes the `spl_tlv_account_resolution` and `spl_transfer_hook_interface` crates to manage extra accounts and enable transfer hooks for SPL tokens.

```
use anchor_lang::{
    prelude::*,
    system_program::{create_account, CreateAccount},
};
use anchor_spl::{
    associated_token::AssociatedToken,
    token_interface::{Mint, TokenAccount, TokenInterface},
};
use spl_tlv_account_resolution::{
    account::ExtraAccountMeta, seeds::Seed, state::ExtraAccou
};
use spl_transfer_hook_interface::instruction::{ExecuteInstruc

declare_id!("DrWbQtYJGtsoRwzKqAbHKHKsCJJfpysudF39GBVFSxub");

#[error_code]
pub enum MyError {
    #[msg("The amount is too big")]
    AmountTooBig,
}

#[program]
pub mod transfer_hook {
    use super::*;

    pub fn initialize_extra_account_meta_list(
        ctx: Context<InitializeExtraAccountMetaList>,
    ) -> Result<()> {

        let account_metas = vec![
            ExtraAccountMeta::new_with_seeds(
                &[Seed::Literal {
                    bytes: "counter".as_bytes().to_vec(),
                }],
                false, // is_signer
                true,  // is_writable
            )?,
        ];

        // calculate account size
        let account_size = ExtraAccountMetaList::size_of(acco
        // calculate minimum required lamports
        let lamports = Rent::get()?.minimum_balance(account_s

        let mint = ctx.accounts.mint.key();
        let signer_seeds: &[&[&[u8]]] = &[&[
            b"extra-account-metas",
            &mint.as_ref(),
            &[ctx.bumps.extra_account_meta_list],
        ]];

        // create ExtraAccountMetaList account
        create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.extra_account_meta_list.
                },
            )
            .with_signer(signer_seeds),
            lamports,
            account_size,
            ctx.program_id,
        )?;

        // initialize ExtraAccountMetaList account with extra
        ExtraAccountMetaList::init::<ExecuteInstruction>(
            &mut ctx.accounts.extra_account_meta_list.try_bor
            &account_metas,
        )?;
```

```
        Ok(())
    }

    pub fn transfer_hook(ctx: Context<TransferHook>, amount:

        if amount > 50 {
            msg!("The amount is too big {0}", amount);
            //return err!(MyError::AmountTooBig);
        }

        ctx.accounts.counter_account.counter.checked_add(1).u

        msg!("This token has been transferred {0} times", ctx

        Ok(())
    }

    // fallback instruction handler as workaround to anchor i
    pub fn fallback<'info>(
        program_id: &Pubkey,
        accounts: &'info [AccountInfo<'info>],
        data: &[u8],
    ) -> Result<()> {
        let instruction = TransferHookInstruction::unpack(dat

        // match instruction discriminator to transfer hook i
        // token2022 program CPIs this instruction on token t
        match instruction {
            TransferHookInstruction::Execute { amount } => {
                let amount_bytes = amount.to_le_bytes();

                // invoke custom transfer hook instruction on
                __private::__global::transfer_hook(program_id
            }
            _ => return Err(ProgramError::InvalidInstructionD
        }
    }
```

```rust
}

#[derive(Accounts)]
pub struct InitializeExtraAccountMetaList<'info> {
    #[account(mut)]
    payer: Signer<'info>,

    /// CHECK: ExtraAccountMetaList Account, must use these s
    #[account(
        mut,
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: AccountInfo<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        init_if_needed,
        seeds = [b"counter"],
        bump,
        payer = payer,
        space = 16
    )]
    pub counter_account: Account<'info, CounterAccount>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
    pub system_program: Program<'info, System>,
}

// Order of accounts matters for this struct.
// The first 4 accounts are the accounts required for token t
// Remaining accounts are the extra accounts required from the
// These accounts are provided via CPI to this program from t
#[derive(Accounts)]
pub struct TransferHook<'info> {
    #[account(
        token::mint = mint,
        token::authority = owner,
    )]
    pub source_token: InterfaceAccount<'info, TokenAccount>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        token::mint = mint,
    )]
    pub destination_token: InterfaceAccount<'info, TokenAccou
    /// CHECK: source token account owner, can be SystemAccou
    pub owner: UncheckedAccount<'info>,
    /// CHECK: ExtraAccountMetaList Account,
    #[account(
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: UncheckedAccount<'info>,
    #[account(
        seeds = [b"counter"],
        bump
    )]
    pub counter_account: Account<'info, CounterAccount>,
}

#[account]
pub struct CounterAccount {
    counter: u64,
}
```

helloworld-transferhook.rs

This Solana program built with the Anchor framework defines instructions for initializing an extra account metadata list and implementing a custom transfer hook. The `initialize_extra_account_meta_list` instruction creates and initializes an ExtraAccountMetaList account, which can be used to store additional accounts required for specific operations. The `transfer_hook` instruction allows executing custom logic when a token transfer occurs. The program utilizes the `spl_tlv_account_resolution` and `spl_transfer_hook_interface` crates to enable transfer hooks for SPL tokens.

```
use anchor_lang::{
    prelude::*,
    system_program::{create_account, CreateAccount},
};
use anchor_spl::{
    associated_token::AssociatedToken,
    token_interface::{Mint, TokenAccount, TokenInterface},
};
use spl_tlv_account_resolution::{
    state::ExtraAccountMetaList,
};
use spl_transfer_hook_interface::instruction::{ExecuteInstruc

declare_id!("DrWbQtYJGtsoRwzKqAbHKHKsCJJfpysudF39GBVFSxub");

#[program]
pub mod transfer_hook {
    use super::*;

    pub fn initialize_extra_account_meta_list(
        ctx: Context<InitializeExtraAccountMetaList>,
    ) -> Result<()> {

        let account_metas = vec![];

        // calculate account size
        let account_size = ExtraAccountMetaList::size_of(acco
        // calculate minimum required lamports
        let lamports = Rent::get()?.minimum_balance(account_s

        let mint = ctx.accounts.mint.key();
        let signer_seeds: &[&[&[u8]]] = &[&[
            b"extra-account-metas",
            &mint.as_ref(),
            &[ctx.bumps.extra_account_meta_list],
        ]];

        // create ExtraAccountMetaList account
        create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.extra_account_meta_list.
                },
            )
            .with_signer(signer_seeds),
            lamports,
            account_size,
            ctx.program_id,
        )?;

        // initialize ExtraAccountMetaList account with extra
        ExtraAccountMetaList::init::<ExecuteInstruction>(
            &mut ctx.accounts.extra_account_meta_list.try_bor
            &account_metas,
        )?;

        Ok(())
    }

    pub fn transfer_hook(ctx: Context<TransferHook>, amount:

        msg!("Hello Transfer Hook!");

        Ok(())
    }

    // fallback instruction handler as workaround to anchor i
    pub fn fallback<'info>(
        program_id: &Pubkey,
        accounts: &'info [AccountInfo<'info>],
        data: &[u8],
    ) -> Result<()> {
        let instruction = TransferHookInstruction::unpack(dat

        // match instruction discriminator to transfer hook i
        // token2022 program CPIs this instruction on token t
        match instruction {
            TransferHookInstruction::Execute { amount } => {
                let amount_bytes = amount.to_le_bytes();

                // invoke custom transfer hook instruction on
                __private::__global::transfer_hook(program_id
            }
            _ => return Err(ProgramError::InvalidInstructionD
        }
    }
}

#[derive(Accounts)]
pub struct InitializeExtraAccountMetaList<'info> {
    #[account(mut)]
    payer: Signer<'info>,

    /// CHECK: ExtraAccountMetaList Account, must use these s
    #[account(
        mut,
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: AccountInfo<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
    pub system_program: Program<'info, System>,
}

// Order of accounts matters for this struct.
// The first 4 accounts are the accounts required for token t
// Remaining accounts are the extra accounts required from th
// These accounts are provided via CPI to this program from t
#[derive(Accounts)]
pub struct TransferHook<'info> {
    #[account(
        token::mint = mint,
        token::authority = owner,
    )]
    pub source_token: InterfaceAccount<'info, TokenAccount>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        token::mint = mint,
    )]
    pub destination_token: InterfaceAccount<'info, TokenAccou
    /// CHECK: source token account owner, can be SystemAccou
    pub owner: UncheckedAccount<'info>,
    /// CHECK: ExtraAccountMetaList Account,
    #[account(
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: UncheckedAccount<'info>,
}
```

TransferHook-TransferCost.rs

This Solana program built with the Anchor framework defines instructions for initializing an extra account metadata list and implementing a custom transfer hook that charges a SOL fee on token transfers. It utilizes the `spl_tlv_account_resolution` and `spl_transfer_hook_interface` crates to manage extra accounts and enable transfer hooks for SPL tokens. The transfer hook instruction transfers a portion of the wrapped SOL (WSOL) tokens from the sender's account to a delegate account as a fee for the token transfer.

```
use anchor_lang::{
    prelude::*,
    system_program::{create_account, CreateAccount},
};
use anchor_spl::{
    associated_token::AssociatedToken,
    token_interface::{transfer_checked, Mint, TokenAccount, T
};
use spl_tlv_account_resolution::{
    account::ExtraAccountMeta, seeds::Seed, state::ExtraAccou
};
```

```
use spl_transfer_hook_interface::instruction::{ExecuteInstruc

// transfer-hook program that charges a SOL fee on token tran
// use a delegate and wrapped SOL because signers from initia

declare_id!("DrWbQtYJGtsoRwzKqAbHKHKsCJJfpysudF39GBVFSxub");

#[error_code]
pub enum MyError {
   #[msg("Amount Too big")]
   AmountTooBig,
}

#[program]
pub mod transfer_hook {
    use super::*;

    pub fn initialize_extra_account_meta_list(
        ctx: Context<InitializeExtraAccountMetaList>,
    ) -> Result<()> {
        // index 0-3 are the accounts required for token tran
        // index 4 is address of ExtraAccountMetaList account
        let account_metas = vec![
            // index 5, wrapped SOL mint
            ExtraAccountMeta::new_with_pubkey(&ctx.accounts.w
            // index 6, token program
            ExtraAccountMeta::new_with_pubkey(&ctx.accounts.t
            // index 7, associated token program
            ExtraAccountMeta::new_with_pubkey(
                &ctx.accounts.associated_token_program.key(),
                false,
                false,
            )?,
            // index 8, delegate PDA
            ExtraAccountMeta::new_with_seeds(
                &[Seed::Literal {
                    bytes: "delegate".as_bytes().to_vec(),
                }],
                false, // is_signer
                true,  // is_writable
            )?,
            // index 9, delegate wrapped SOL token account
            ExtraAccountMeta::new_external_pda_with_seeds(
                7, // associated token program index
                &[
                    Seed::AccountKey { index: 8 }, // owner i
                    Seed::AccountKey { index: 6 }, // token p
                    Seed::AccountKey { index: 5 }, // wsol mi
                ],
                false, // is_signer
                true,  // is_writable
            )?,
            // index 10, sender wrapped SOL token account
            ExtraAccountMeta::new_external_pda_with_seeds(
                7, // associated token program index
                &[
                    Seed::AccountKey { index: 3 }, // owner i
                    Seed::AccountKey { index: 6 }, // token p
                    Seed::AccountKey { index: 5 }, // wsol mi
                ],
                false, // is_signer
                true,  // is_writable
            )?,
            ExtraAccountMeta::new_with_seeds(
                &[Seed::Literal {
                    bytes: "counter".as_bytes().to_vec(),
                }],
                false, // is_signer
                true,  // is_writable
            )?,
        ];

        // calculate account size
        let account_size = ExtraAccountMetaList::size_of(acco
        // calculate minimum required lamports
        let lamports = Rent::get()?.minimum_balance(account_s
        let mint = ctx.accounts.mint.key();
        let signer_seeds: &[&[&[u8]]] = &[&[
            b"extra-account-metas",
            &mint.as_ref(),
            &[ctx.bumps.extra_account_meta_list],
        ]];

        // create ExtraAccountMetaList account
        create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.extra_account_meta_list.
                },
            )
            .with_signer(signer_seeds),
            lamports,
            account_size,
            ctx.program_id,
        )?;

        // initialize ExtraAccountMetaList account with extra
        ExtraAccountMetaList::init::<ExecuteInstruction>(
            &mut ctx.accounts.extra_account_meta_list.try_bor
            &account_metas,
        )?;

        Ok(())
    }

    pub fn transfer_hook(ctx: Context<TransferHook>, amount:

        if amount > 50 {
            //msg!("The amount is too big {0}", amount);
            //return err!(MyError::AmountTooBig);
        }
        ctx.accounts.counter_account.counter += 1;

        msg!("This token has been transferred {0} times", ctx

        // All accounts are non writable so you can not burn 
        msg!("Is writable mint {0}", ctx.accounts.mint.to_acc
        msg!("Is destination mint {0}", ctx.accounts.destinat
        msg!("Is source mint {0}", ctx.accounts.source_token.

        let signer_seeds: &[&[&[u8]]] = &[&[b"delegate", &[ct

        // Transfer WSOL from sender to delegate token accoun
        transfer_checked(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                TransferChecked {
                    from: ctx.accounts.sender_wsol_token_acco
                    mint: ctx.accounts.wsol_mint.to_account_i
                    to: ctx.accounts.delegate_wsol_token_acco
                    authority: ctx.accounts.delegate.to_accou
                },
            )
            .with_signer(signer_seeds),
            amount / 2,
            ctx.accounts.wsol_mint.decimals,
        )?;
        Ok(())
    }

    // fallback instruction handler as workaround to anchor i
    pub fn fallback<'info>(
        program_id: &Pubkey,
        accounts: &'info [AccountInfo<'info>],
        data: &[u8],
    ) -> Result<()> {
        let instruction = TransferHookInstruction::unpack(dat
        // match instruction discriminator to transfer hook i
        // token2022 program CPIs this instruction on token t
        match instruction {
            TransferHookInstruction::Execute { amount } => {
                let amount_bytes = amount.to_le_bytes();

                // invoke custom transfer hook instruction on
                __private::__global::transfer_hook(program_id
            }
            _ => Err(ProgramError::InvalidInstructionData.int
        }
    }
}

#[derive(Accounts)]
pub struct InitializeExtraAccountMetaList<'info> {
    #[account(mut)]
    payer: Signer<'info>,

    /// CHECK: ExtraAccountMetaList Account, must use these s
    #[account(
        mut,
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: AccountInfo<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    pub wsol_mint: InterfaceAccount<'info, Mint>,
    #[account(
        init,
        seeds = [b"counter"],
        bump,
        payer = payer,
        space = 9
    )]
    pub counter_account: Account<'info, CounterAccount>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
    pub system_program: Program<'info, System>,
}

// Order of accounts matters for this struct.
// The first 4 accounts are the accounts required for token t
// Remaining accounts are the extra accounts required from th
// These accounts are provided via CPI to this program from t
#[derive(Accounts)]
pub struct TransferHook<'info> {
    #[account(
        token::mint = mint,
        token::authority = owner,
    )]
    pub source_token: InterfaceAccount<'info, TokenAccount>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        token::mint = mint,
    )]
    pub destination_token: InterfaceAccount<'info, TokenAccou
    /// CHECK: source token account owner, can be SystemAccou
    pub owner: UncheckedAccount<'info>,
    /// CHECK: ExtraAccountMetaList Account,
    #[account(
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: UncheckedAccount<'info>,
    pub wsol_mint: InterfaceAccount<'info, Mint>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
    #[account(
        mut,
        seeds = [b"delegate"],
        bump
    )]
    pub delegate: SystemAccount<'info>,
    #[account(
        mut,
        token::mint = wsol_mint,
        token::authority = delegate,
    )]
    pub delegate_wsol_token_account: InterfaceAccount<'info, 
    #[account(
        mut,
        token::mint = wsol_mint,
        token::authority = owner,
    )]
    pub sender_wsol_token_account: InterfaceAccount<'info, To
    #[account(
        seeds = [b"counter"],
        bump
    )]
    pub counter_account: Account<'info, CounterAccount>,
}

#[account]
pub struct CounterAccount {
    counter: u8
}
```

TransferHook-TransferWhitelist.rs

This Solana program built with the Anchor framework implements a whitelist feature for token transfers. It defines instructions for initializing an extra account metadata list, performing a transfer hook that checks if the destination account is whitelisted, and adding new accounts to the whitelist. The program utilizes the `spl_tlv_account_resolution` and `spl_transfer_hook_interface` crates to manage extra accounts and enable transfer hooks for SPL tokens. The whitelist is stored in a dedicated account, and only the authority can add new accounts to the whitelist.

```
use anchor_lang::{
    prelude::*,
    system_program::{create_account, CreateAccount},
};
use anchor_spl::{
    associated_token::AssociatedToken, token_interface::{Mint
```

```
};
use spl_tlv_account_resolution::state::ExtraAccountMetaList;
use spl_transfer_hook_interface::instruction::{ExecuteInstruc

declare_id!("DrWbQtYJGtsoRwzKqAbHKHKsCJJfpysudF39GBVFSxub");

#[program]
pub mod transfer_hook {

    use spl_tlv_account_resolution::{account::ExtraAccountMet

    use super::*;

    pub fn initialize_extra_account_meta_list(
        ctx: Context<InitializeExtraAccountMetaList>,
    ) -> Result<()> {

        let account_metas = vec![
            ExtraAccountMeta::new_with_seeds(
                &[Seed::Literal {
                    bytes: "white_list".as_bytes().to_vec(),
                }], // owner index (delegate PDA)
                false, // is_signer
                true,  // is_writable
            )?,
        ];

        ctx.accounts.white_list.authority = ctx.accounts.paye

        // calculate account size
        let account_size = ExtraAccountMetaList::size_of(acco
        // calculate minimum required lamports
        let lamports = Rent::get()?.minimum_balance(account_s

        let mint = ctx.accounts.mint.key();
        let signer_seeds: &[&[&[u8]]] = &[&[
            b"extra-account-metas",
            &mint.as_ref(),
            &[ctx.bumps.extra_account_meta_list],
        ]];

        // create ExtraAccountMetaList account
        create_account(
            CpiContext::new(
                ctx.accounts.system_program.to_account_info()
                CreateAccount {
                    from: ctx.accounts.payer.to_account_info(
                    to: ctx.accounts.extra_account_meta_list.
                },
            )
            .with_signer(signer_seeds),
            lamports,
            account_size,
            ctx.program_id,
        )?;

        // initialize ExtraAccountMetaList account with extra
        ExtraAccountMetaList::init::<ExecuteInstruction>(
            &mut ctx.accounts.extra_account_meta_list.try_bor
            &account_metas,
        )?;

        Ok(())
    }

    pub fn transfer_hook(ctx: Context<TransferHook>, _amount:

        if !ctx.accounts.white_list.white_list.contains(&ctx.
            panic!("Account not in white list!");
        }

        msg!("Account in white list, all good!");

        Ok(())
    }
    pub fn add_to_whitelist(ctx: Context<AddToWhiteList>) ->

        if ctx.accounts.white_list.authority != ctx.accounts.
            panic!("Only the authority can add to the white l
        }

        ctx.accounts.white_list.white_list.push(ctx.accounts.
        msg!("New account white listed! {0}", ctx.accounts.ne
        msg!("White list length! {0}", ctx.accounts.white_lis

        Ok(())
    }

    // fallback instruction handler as workaround to anchor i
    pub fn fallback<'info>(
        program_id: &Pubkey,
        accounts: &'info [AccountInfo<'info>],
        data: &[u8],
    ) -> Result<()> {
        let instruction = TransferHookInstruction::unpack(dat

        // match instruction discriminator to transfer hook i
        // token2022 program CPIs this instruction on token t
        match instruction {
            TransferHookInstruction::Execute { amount } => {
                let amount_bytes = amount.to_le_bytes();

                // invoke custom transfer hook instruction on
                __private::__global::transfer_hook(program_id
            }
            _ => return Err(ProgramError::InvalidInstructionD
        }
    }
}

#[derive(Accounts)]
pub struct InitializeExtraAccountMetaList<'info> {
    #[account(mut)]
    payer: Signer<'info>,

    /// CHECK: ExtraAccountMetaList Account, must use these s
    #[account(
        mut,
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: AccountInfo<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    pub token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedTo
    pub system_program: Program<'info, System>,
    #[account(
        init_if_needed,
        seeds = [b"white_list"],
        bump,
        payer = payer,
        space = 400
    )]
    pub white_list: Account<'info, WhiteList>,
}

// Order of accounts matters for this struct.
// The first 4 accounts are the accounts required for token t
// Remaining accounts are the extra accounts required from th
// These accounts are provided via CPI to this program from t
#[derive(Accounts)]
pub struct TransferHook<'info> {
    #[account(
        token::mint = mint,
        token::authority = owner,
    )]
    pub source_token: InterfaceAccount<'info, TokenAccount>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        token::mint = mint,
    )]
    pub destination_token: InterfaceAccount<'info, TokenAccou
    /// CHECK: source token account owner, can be SystemAccou
    pub owner: UncheckedAccount<'info>,
    /// CHECK: ExtraAccountMetaList Account,
    #[account(
        seeds = [b"extra-account-metas", mint.key().as_ref()]
        bump
    )]
    pub extra_account_meta_list: UncheckedAccount<'info>,
    #[account(
        seeds = [b"white_list"],
        bump
    )]
    pub white_list: Account<'info, WhiteList>,
}

#[derive(Accounts)]
pub struct AddToWhiteList<'info> {
    /// CHECK: New account to add to white list
    #[account()]
    pub new_account: AccountInfo<'info>,
    #[account(
        mut,
        seeds = [b"white_list"],
        bump
    )]
    pub white_list: Account<'info, WhiteList>,
    #[account(mut)]
    pub signer: Signer<'info>,
}

#[account]
pub struct WhiteList {
    pub authority: Pubkey,
    pub white_list: Vec<Pubkey>,
}
```

TokenSwap.rs

This code defines the main entry point for a Solana program built using the Anchor framework. The program is designed to facilitate decentralized token swaps using an Automated Market Maker (AMM) model. It includes several instructions for creating an AMM, creating a liquidity pool, depositing liquidity into the pool, withdrawing liquidity from the pool, and swapping tokens. The program likely utilizes other modules for defining constants, error handling, and managing the state of the AMM and liquidity pool.

```rust
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

mod constants;
mod errors;
mod instructions;
mod state;

// Set the correct key here
declare_id!("C3ti6PFK6PoYShRFx1BNNTQU3qeY1iVwjwCA6SjJhiuW");

#[program]
pub mod swap_example {
    pub use super::instructions::*;
    use super::*;

    pub fn create_amm(ctx: Context<CreateAmm>, id: Pubkey, fe
        instructions::create_amm(ctx, id, fee)
    }

    pub fn create_pool(ctx: Context<CreatePool>) -> Result<()
        instructions::create_pool(ctx)
    }

    pub fn deposit_liquidity(
        ctx: Context<DepositLiquidity>,
        amount_a: u64,
        amount_b: u64,
    ) -> Result<()> {
        instructions::deposit_liquidity(ctx, amount_a, amount_
    }

    pub fn withdraw_liquidity(ctx: Context<WithdrawLiquidity>
        instructions::withdraw_liquidity(ctx, amount)
    }

    pub fn swap_exact_tokens_for_tokens(
        ctx: Context<SwapExactTokensForTokens>,
        swap_a: bool,
        input_amount: u64,
        min_output_amount: u64,
    ) -> Result<()> {
        instructions::swap_exact_tokens_for_tokens(ctx, swap_
    }
}
```

TransferToken.rs

Solana program for creating, minting, and transfer of tokens using Anchor.

```
#![allow(clippy::result_large_err)]

use anchor_lang::prelude::*;

pub mod instructions;

use instructions::*;

declare_id!("2W7B8C5skxyVaAA1LfYAsRHiv26LL5j88GJ9XYyybWqc");

#[program]
pub mod transfer_tokens {
    use super::*;

    pub fn create_token(
        ctx: Context<CreateToken>,
        token_title: String,
        token_symbol: String,
        token_uri: String,
    ) -> Result<()> {
        create::create_token(ctx, token_title, token_symbol,
    }

    pub fn mint_token(ctx: Context<MintToken>, amount: u64) -
        mint::mint_token(ctx, amount)
    }

    pub fn transfer_tokens(ctx: Context<TransferTokens>, amou
        transfer::transfer_tokens(ctx, amount)
    }
}
```

fetchNFTsByGroups.ts

This script demonstrates how to use the Metaplex Read API to fetch compressed NFTs (Non-Fungible Tokens) associated with a specific address. It loads public keys from a file, establishes a connection to the Solana cluster, and retrieves a list of assets. The script then processes the response, logs relevant information about each asset, and saves the asset IDs to a file

```
/**
 * Demonstrate the use of a few of the Metaplex Read API meth
 * (needed to fetch compressed NFTs)
 * using the `@metaplex-foundation/js` sdk
 */

// import custom helpers for demos
import {
  loadPublicKeysFromFile,
  printConsoleSeparator,
  savePublicKeyToFile,
} from "@/utils/helpers";

// imports from other libraries
import dotenv from "dotenv";
import { Metaplex, MetaplexError, ReadApiAssetList } from "@m
import { ReadApiConnection } from "@metaplex-foundation/js";
import { PublicKey } from "@solana/web3.js";

// load the env variables and store the cluster RPC url
dotenv.config();
const CLUSTER_URL = process.env.RPC_URL ?? "";

(async () => {
  // load the stored PublicKeys for ease of use
  let keys = loadPublicKeysFromFile();

  // ensure the primary script was already run
  if (!keys?.collectionMint || !keys?.treeAddress)
    return console.warn("No local keys were found. Please run

  const treeAddress: PublicKey = keys.treeAddress;
  const treeAuthority: PublicKey = keys.treeAuthority;
  const collectionMint: PublicKey = keys.collectionMint;
  const userAddress: PublicKey = keys.userAddress;
  const testWallet: PublicKey = keys.testWallet;

  console.log("==== Local PublicKeys loaded ====");
  console.log("Tree address:", treeAddress.toBase58());
  console.log("Tree authority:", treeAuthority.toBase58());
  console.log("Collection mint:", collectionMint.toBase58());
  console.log("User address:", userAddress.toBase58());
  console.log("Test address:", testWallet.toBase58());

  ///////////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////////

  // define the address we are actually going to check (in th
  const checkAddress = collectionMint.toBase58();

  printConsoleSeparator(`getAssetsByGroup: ${checkAddress}`);

  const connection = new ReadApiConnection(CLUSTER_URL);
  const metaplex = Metaplex.make(connection);
```

```
/**
 * Fetch a listing of NFT assets by an owner's address (via
 * ---
 * NOTE: This will return both compressed NFTs AND traditio
 */
const rpcAssets = await metaplex
  .rpc()
  .getAssetsByGroup({
    groupKey: "collection",
    groupValue: checkAddress,
    sortBy: {
      sortBy: "created",
      sortDirection: "asc",
    },
  })
  .then(res => {
    if ((res as MetaplexError)?.cause) throw res;
    else return res as ReadApiAssetList;
  });

/**
 * Process the returned `rpcAssets` response
 */
console.log("Total assets returned:", rpcAssets.total);

// loop over each of the asset items in the collection
rpcAssets.items.map(asset => {
  // only show compressed nft assets
  if (!asset.compression.compressed) return;

  // display a spacer between each of the assets
  console.log("\n=========================================

  // locally save the addresses for the demo
  savePublicKeyToFile("assetIdTestAddress", new PublicKey(a

  // extra useful info
  console.log("assetId:", asset.id);
```

```
    // view the ownership info for the given asset
    console.log("ownership:", asset.ownership);

    // metadata json data (auto fetched thanks to the Metaple
    // console.log("metadata:", asset.content.metadata);

    // view the compression specific data for the given asset
    console.log("compression:", asset.compression);

    if (asset.compression.compressed) {
      console.log("==> This NFT is compressed! <===");
      console.log("\tleaf_id:", asset.compression.leaf_id);
    } else console.log("==> NFT is NOT compressed! <===");
  });
})();
```

fetchNFTByOwner.ts

This script demonstrates how to use the Metaplex Read API to fetch compressed NFTs (Non-Fungible Tokens) associated with a specific address. It loads public keys from a file, and retrieves a list of assets from the specific wallet.

```
/**
 * Demonstrate the use of a few of the Metaplex Read API meth
 * (needed to fetch compressed NFTs)
 * using the `@metaplex-foundation/js` sdk
 */

// import custom helpers for demos
import {
  loadPublicKeysFromFile,
  printConsoleSeparator,
  savePublicKeyToFile,
} from "@/utils/helpers";

// imports from other libraries
import dotenv from "dotenv";
import { Metaplex, MetaplexError, ReadApiAssetList } from "@m
import { ReadApiConnection } from "@metaplex-foundation/js";
import { PublicKey } from "@solana/web3.js";

// load the env variables and store the cluster RPC url
dotenv.config();
const CLUSTER_URL = process.env.RPC_URL ?? "";

(async () => {
  // load the stored PublicKeys for ease of use
  let keys = loadPublicKeysFromFile();

  // ensure the primary script was already run
  if (!keys?.collectionMint || !keys?.treeAddress)
    return console.warn("No local keys were found. Please run

  const treeAddress: PublicKey = keys.treeAddress;
  const treeAuthority: PublicKey = keys.treeAuthority;
  const collectionMint: PublicKey = keys.collectionMint;
  const userAddress: PublicKey = keys.userAddress;
  const testWallet: PublicKey = keys.testWallet;

  console.log("==== Local PublicKeys loaded ====");
  console.log("Tree address:", treeAddress.toBase58());
  console.log("Tree authority:", treeAuthority.toBase58());
  console.log("Collection mint:", collectionMint.toBase58());
  console.log("User address:", userAddress.toBase58());
  console.log("Test address:", testWallet.toBase58());

  // define the address we are actually going to check (in th
  // const checkAddress = testWallet.toBase58();
  const checkAddress = userAddress.toBase58();

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  printConsoleSeparator(`getAssetsByOwner: ${checkAddress}`);
```

```
const connection = new ReadApiConnection(CLUSTER_URL);
const metaplex = Metaplex.make(connection);

/**
 * Fetch a listing of NFT assets by an owner's address (via
 * ---
 * NOTE: This will return both compressed NFTs AND traditio
 */
const rpcAssets = await metaplex
  .rpc()
  .getAssetsByOwner({
    ownerAddress: checkAddress,
  })
  .then(res => {
    if ((res as MetaplexError)?.cause) throw res;
    else return res as ReadApiAssetList;
  });

/**
 * Process the returned `rpcAssets` response
 */
console.log("Total assets returned:", rpcAssets.total);

// loop over each of the asset items in the collection
rpcAssets.items.map(asset => {
  // only show compressed nft assets
  if (!asset.compression.compressed) return;

  // display a spacer between each of the assets
  console.log("\n=========================================

  // locally save the addresses for the demo
  savePublicKeyToFile("assetIdTestAddress", new PublicKey(a

  // extra useful info
  console.log("assetId:", asset.id);

  // view the ownership info for the given asset
```

```
    console.log("ownership:", asset.ownership);

    // metadata json data (auto fetched thanks to the Metaple
    // console.log("metadata:", asset.content.metadata);

    // view the compression specific data for the given asset
    console.log("compression:", asset.compression);

    if (asset.compression.compressed) {
      console.log("==> This NFT is compressed! <===");
      console.log("\tleaf_id:", asset.compression.leaf_id);
    } else console.log("==> NFT is NOT compressed! <===");
  });
})();
```

mintToCollection.ts

This script demonstrates how to mint an additional compressed NFT to an existing tree and/or collection, using the `@metaplex-foundation/js` SDK

```
/**
 * This script demonstrates how to mint an additional compres
 * existing tree and/or collection, using the `@metaplex-foun
 * ---
 * NOTE: A collection can use multiple trees to store compres
 * This example uses the same tree for simplicity.
 */

// imports from other libraries
import dotenv from "dotenv";
import { Metaplex, ReadApiConnection, keypairIdentity } from
import { PublicKey, clusterApiUrl } from "@solana/web3.js";

// import custom helpers for demos
import {
  loadPublicKeysFromFile,
  loadKeypairFromFile,
  loadOrGenerateKeypair,
  explorerURL,
  printConsoleSeparator,
  savePublicKeyToFile,
} from "@/utils/helpers";
import { getLeafAssetId, metadataArgsBeet } from "@metaplex-f
import {
  changeLogEventV1Beet,
  deserializeApplicationDataEvent,
  deserializeChangeLogEventV1,
} from "@solana/spl-account-compression";
import { bs58 } from "@project-serum/anchor/dist/cjs/utils/by
import { BN } from "@project-serum/anchor";

// load the env variables and store the cluster RPC url
dotenv.config();
const CLUSTER_URL = process.env.RPC_URL ?? clusterApiUrl("dev

// create a new rpc connection
// const connection = new Connection(CLUSTER_URL);
const connection = new ReadApiConnection(CLUSTER_URL);

(async () => {
  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  // generate a new Keypair for testing, named `testWallet`
  const testWallet = loadOrGenerateKeypair("testWallet");

  // generate a new keypair for use in this demo (or load it
  const payer = process.env?.LOCAL_PAYER_JSON_ABSPATH
    ? loadKeypairFromFile(process.env?.LOCAL_PAYER_JSON_ABSPA
    : loadOrGenerateKeypair("payer");

  console.log("Payer address:", payer.publicKey.toBase58());
  console.log("Test wallet address:", testWallet.publicKey.to

  // load the stored PublicKeys for ease of use
  let keys = loadPublicKeysFromFile();
  // ensure the primary script was already run
  if (!keys?.collectionMint || !keys?.treeAddress)
    return console.warn("No local keys were found. Please run

  const treeAddress: PublicKey = keys.treeAddress;
  const collectionMint: PublicKey = keys.collectionMint;
  const collectionAuthority: PublicKey = keys.collectionAutho

  console.log("==== Local PublicKeys loaded ====");
  console.log("Tree address:", treeAddress.toBase58());
  console.log("Collection mint:", collectionMint.toBase58());
  console.log("User address:", payer.publicKey.toBase58());
  console.log("Test address:", testWallet.publicKey.toBase58(

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  // initialize metaplex with our RPC connection, and the pay
  const metaplex = Metaplex.make(connection).use(keypairIdent
  // const metaplex = Metaplex.make(connection).use(keypairId

  printConsoleSeparator("Mint a single compressed NFT into th

  /**
   * minting compressed NFTs with the metaplex sdk is very mu
   *
   * the difference is that when you want to mint a compresse
   * you simply provide the `tree` address that will store th
   *
   * when `tree` is present, the metaplex sdk knows you want
   */

  // mint a new compressed NFT into our existing collection
  const { response, nft } = await metaplex.nfts().create({
    uri: "https://supersweetcollection.notarealurl/token.json
    name: "compressed with metaplex",
    sellerFeeBasisPoints: 500,
    collection: collectionMint,
    // note: the `payer` is also this collection's authority
    collectionAuthority: payer,

    // note: this merkle tree must have already been created
    tree: treeAddress,
  });

  // save the `assetId` of the new compressed NFT locally
  savePublicKeyToFile("assetIdTestAddress", new PublicKey(nft

  console.log("nft minted with metaplex sdk:", nft);

  printConsoleSeparator("View on explorer");

  console.log(explorerURL({ txSignature: response.signature }
})();
```

simpleProofVerification.ts

This script demonstrates how to perform a simple client side proof verification, using data provided from the `@metaplex-foundation/js` SDK

```
/**
 * Demonstrate how to perform a simple client side proof veri
 * data provided from the `@metaplex-foundation/js` SDK
 */

// import custom helpers for demos
import { loadPublicKeysFromFile, printConsoleSeparator } from

import dotenv from "dotenv";
import { GetAssetProofRpcResponse, Metaplex, ReadApiConnectio

// imports from other libraries
import { PublicKey, clusterApiUrl } from "@solana/web3.js";
import {
  ConcurrentMerkleTreeAccount,
  MerkleTree,
  MerkleTreeProof,
} from "@solana/spl-account-compression";

// load the env variables and store the cluster RPC url
dotenv.config();
const CLUSTER_URL = process.env.RPC_URL ?? clusterApiUrl("dev

// create a new rpc connection
// const connection = new Connection(CLUSTER_URL);
const connection = new ReadApiConnection(CLUSTER_URL);

(async () => {
  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  // load the stored PublicKeys for ease of use
  let keys = loadPublicKeysFromFile();

  // ensure the primary script was already run
  if (!keys?.assetIdTestAddress)
    return console.warn(
      "No locally saved `assetIdTestAddress` was found, Pleas
    );

  const assetIdTestAddress: PublicKey = keys.assetIdTestAddre
  const assetIdUserAddress: PublicKey = keys.assetIdUserAddre

  console.log("==== Local PublicKeys loaded ====");
  console.log("Test Asset ID:", assetIdTestAddress.toBase58()
  console.log("User Asset ID:", assetIdUserAddress.toBase58()

  // set the asset to test with
  const assetId = assetIdTestAddress;
  // const assetId = assetIdUserAddress;

  const metaplex = Metaplex.make(connection);

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////
```

```
printConsoleSeparator("Get the compressed nft by its assetI

/**
 * Fetch an asset from the ReadApi by its `assetId`
 */
const nft = await metaplex.nfts().findByAssetId({ assetId }
console.log(nft);

printConsoleSeparator("Get the asset proof from the RPC:");

/**
 * Perform client side verification of the proof that was p
 * ---
 * NOTE: This is not required to be performed, but may aid 
 * due to your RPC providing stale or incorrect data (often
 * The actual proof validation is performed on-chain.
 * ---
 * NOTE: This client side validation is also handled by the
 * transferring a compressed NFT. But it is also show here 
 */

// fetch an asset's proof from the ReadApi by its `assetId`
const assetProof = (await metaplex.rpc().getAssetProof(asse
console.log(assetProof);

// construct a valid proof object to check against
const merkleTreeProof: MerkleTreeProof = {
  leafIndex: nft.compression?.leaf_id || 0,
  leaf: new PublicKey(assetProof.leaf).toBuffer(),
  root: new PublicKey(assetProof.root).toBuffer(),
  proof: assetProof.proof.map((node: string) => new PublicK
};

/**
 * note:
 * the `merkleTreeProof.proof` value is the COMPLETE list o
 * The entire list of these "proof hashes" are required to 
```

```
   * client side verification check. Since this client side c
   * the proof hashes that are stored on chain in the tree's 
   *
   * warning:
   * This is different than when you are sending proof hashes
   * In that case, sending the "complete proof hash list" wil
   * This is because the on-chain program will use ALL the pr
   * (via the `anchorRemainingAccounts` field) to compute the
   *
   */

  printConsoleSeparator("Client side checks of the RPC provid

  // get the actual merkle tree data from the Solana blockcha
  const merkleTree = new PublicKey(assetProof.tree_id);
  const treeAccount = await ConcurrentMerkleTreeAccount.fromA

  const currentRoot = treeAccount.getCurrentRoot();
  const rootFromRpc = new PublicKey(assetProof.root).toBuffer

  console.log("Is RPC provided proof/root valid:", MerkleTree

  /**
   * note: the current on-chain root hash (`currentRoot`) doe
   * RPC provided root hash (`rootFromRpc`). This is because 
   * of valid root hashes are stored on-chain via the trees o
   * (set by your tree's `maxBufferSize` at tree creation)
   *
   * This check is show here purely for demonstration, and is
   */
  console.log(
    "Does the current on-chain root match RPC provided root:"
    new PublicKey(currentRoot).toBase58() === new PublicKey(r
  );
})();
```

transferNFT.ts

Demonstrate how to transfer a compressed NFT from one owner to another, using the `@metaplex-foundation/js` SDK

```
/**
 * Demonstrate how to transfer a compressed NFT from one owne
 * using the `@metaplex-foundation/js` SDK
 */

// imports from other libraries
import dotenv from "dotenv";
import { Metaplex, ReadApiConnection, keypairIdentity } from
import { PublicKey, clusterApiUrl } from "@solana/web3.js";

// import custom helpers for demos
import {
  loadPublicKeysFromFile,
  loadKeypairFromFile,
  loadOrGenerateKeypair,
  explorerURL,
  printConsoleSeparator,
} from "@/utils/helpers";

// load the env variables and store the cluster RPC url
dotenv.config();
const CLUSTER_URL = process.env.RPC_URL ?? clusterApiUrl("dev

// create a new rpc connection
// const connection = new Connection(CLUSTER_URL);
const connection = new ReadApiConnection(CLUSTER_URL);

(async () => {
  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  // generate a new Keypair for testing, named `testWallet`
  const testWallet = loadOrGenerateKeypair("testWallet");

  // generate a new keypair for use in this demo (or load it
  const payer = process.env?.LOCAL_PAYER_JSON_ABSPATH
    ? loadKeypairFromFile(process.env?.LOCAL_PAYER_JSON_ABSPAT
    : loadOrGenerateKeypair("payer");

  console.log("Payer address:", payer.publicKey.toBase58());
  console.log("Test wallet address:", testWallet.publicKey.to

  // load the stored PublicKeys for ease of use
  let keys = loadPublicKeysFromFile();

  // ensure the primary script was already run
  if (!keys?.assetIdTestAddress)
    return console.warn(
      "No locally saved `assetIdTestAddress` was found, Pleas
    );

  const assetIdTestAddress: PublicKey = keys.assetIdTestAddre
  const assetIdUserAddress: PublicKey = keys.assetIdUserAddre

  console.log("==== Local PublicKeys loaded ====");
  console.log("Test Asset ID:", assetIdTestAddress.toBase58()
  console.log("User Asset ID:", assetIdUserAddress.toBase58()

  // set the asset to test with
  const assetId = assetIdTestAddress;
  // const assetId = assetIdUserAddress;

  const metaplex = Metaplex.make(connection).use(keypairIdent
  // const metaplex = Metaplex.make(connection).use(keypairId

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  printConsoleSeparator("Get the compressed nft by its assetI

  /**
   * Fetch an asset from the ReadApi by its `assetId`
   */
```

```
    const nft = await metaplex.nfts().findByAssetId({ assetId }
    console.log(nft);

    /**
     * Use the Metaplex SDK to perform the transfer of a compre
     * ---
     * NOTE: When your `nftOrSft` was already retrieved using t
     * (via `metaplex.nfts().findByAssetId()`), the Metaplex SD
     * auto-magically handle the rest of the data fetching (inc
     * proof and merkle tree), * as well as perform some client
     * that the RPC provided proof is valid to complete the tra
     */

    printConsoleSeparator("Transfer the compressed nft:");

    await metaplex
      .nfts()
      .transfer({
        nftOrSft: nft,
        toOwner: payer.publicKey,
      })
      .then(res => {
        console.log("transfer complete:", res);

        console.log(explorerURL({ txSignature: res.response.sig
      })
      .catch(err => {
        console.log("==================");
        console.log("  Transfer failed!");
        console.log("==================");
        console.error(err);
      });
  })();
```

changeLog.ts

This script demonstrates the use of a few of the Metaplex Read API methods.

```
/**
 * Demonstrate the use of a few of the Metaplex Read API meth
 * (needed to fetch compressed NFTs)
 */

// local import of the connection wrapper, to help with using
import { WrapperConnection } from "@/ReadApi/WrapperConnectio

// import custom helpers for demos
import {
  explorerURL,
  loadKeypairFromFile,
  loadOrGenerateKeypair,
  loadPublicKeysFromFile,
  printConsoleSeparator,
} from "@/utils/helpers";
import {
  TokenProgramVersion,
  TokenStandard,
  computeCreatorHash,
  computeDataHash,
  createVerifyCreatorInstruction,
  getLeafAssetId,
} from "@metaplex-foundation/mpl-bubblegum";

import {
  SPL_ACCOUNT_COMPRESSION_PROGRAM_ID,
  SPL_NOOP_PROGRAM_ID,
  ConcurrentMerkleTreeAccount,
  getAllChangeLogEventV1FromTransaction,
} from "@solana/spl-account-compression";
import {
  MetadataArgs,
  Creator,
  PROGRAM_ID as BUBBLEGUM_PROGRAM_ID,
} from "@metaplex-foundation/mpl-bubblegum";
import {
  AccountMeta,
  PublicKey,
  TransactionMessage,
  VersionedTransaction,
  clusterApiUrl,
} from "@solana/web3.js";

import dotenv from "dotenv";
import { BN } from "@project-serum/anchor";
dotenv.config();

(async () => {
  // generate a new Keypair for testing, named `wallet`
  const testWallet = loadOrGenerateKeypair("testWallet");

  // generate a new keypair for use in this demo (or load it
  const payer = process.env?.LOCAL_PAYER_JSON_ABSPATH
    ? loadKeypairFromFile(process.env?.LOCAL_PAYER_JSON_ABSPA
    : loadOrGenerateKeypair("payer");

  console.log("Payer address:", payer.publicKey.toBase58());
  console.log("Test wallet address:", testWallet.publicKey.to

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  // load the env variables and store the cluster RPC url
  const CLUSTER_URL = process.env.RPC_URL ?? clusterApiUrl("d

  // create a new rpc connection, using the ReadApi wrapper
  const connection = new WrapperConnection(CLUSTER_URL);

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  printConsoleSeparator("");
  // load the stored PublicKeys for ease of use

  // https://explorer.solana.com/tx/3V1pMta1aKzJdnYwttHR5qtrP
  const devnetSig =
    "3V1pMta1aKzJdnYwttHR5qtrPVQfazwdUykvgM1YY5Xwb2x6hwttLj9R

  const tx = await connection.getTransaction(devnetSig, {
    maxSupportedTransactionVersion: 0,
  });

  if (!tx) throw Error("Tx not found");

  printConsoleSeparator("Events:");

  const events = getAllChangeLogEventV1FromTransaction(tx);

  console.log(events);
  const leafIndex = events[0].index;

  const assetId = await getLeafAssetId(events[0].treeId, new

  console.log("assetId:", assetId);
  console.log("total events:", events.length);
})();
```

createAndMint.ts

Program to create and mint nft collection on chain using Metaplex API.

```
/**
  Overall flow of this script
  - load or create two keypairs (named `payer` and `testWalle
  - create a new tree with enough space to mint all the nft's
  - create a new NFT Collection on chain (using the usual Met
  - mint a single compressed nft into the tree to the `payer`
  - mint a single compressed nft into the tree to the `testWa
  - display the overall cost to perform all these actions
  ---
  NOTE: this script is identical to the `scripts/verboseCreat
  less console logging and explanation of what is occurring
*/

import { Keypair, LAMPORTS_PER_SOL, clusterApiUrl } from "@so
import { ValidDepthSizePair } from "@solana/spl-account-compr
import {
  MetadataArgs,
  TokenProgramVersion,
  TokenStandard,
} from "@metaplex-foundation/mpl-bubblegum";
import { CreateMetadataAccountArgsV3 } from "@metaplex-founda

// import custom helpers for demos
import { loadKeypairFromFile, loadOrGenerateKeypair, numberFo

// import custom helpers to mint compressed NFTs
import { createCollection, createTree, mintCompressedNFT } fr

// local import of the connection wrapper, to help with using
import { WrapperConnection } from "@/ReadApi/WrapperConnectio

import dotenv from "dotenv";
dotenv.config();

// define some reusable balance values for tracking
let initBalance: number, balance: number;

(async () => {
  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  // generate a new Keypair for testing, named `wallet`
  const testWallet = loadOrGenerateKeypair("testWallet");

  // generate a new keypair for use in this demo (or load it
  const payer = process.env?.LOCAL_PAYER_JSON_ABSPATH
```

```
    ? loadKeypairFromFile(process.env?.LOCAL_PAYER_JSON_ABSPAT
    : loadOrGenerateKeypair("payer");

  console.log("Payer address:", payer.publicKey.toBase58());
  console.log("Test wallet address:", testWallet.publicKey.to

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  // load the env variables and store the cluster RPC url
  const CLUSTER_URL = process.env.RPC_URL ?? clusterApiUrl("d

  // create a new rpc connection, using the ReadApi wrapper
  const connection = new WrapperConnection(CLUSTER_URL, "conf

  // get the payer's starting balance (only used for demonstr
  initBalance = await connection.getBalance(payer.publicKey);

  ///////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////

  /*
    Define our tree size parameters
  */
  const maxDepthSizePair: ValidDepthSizePair = {
    // max=16,384 nodes
    maxDepth: 14,
    maxBufferSize: 64,
  };
  const canopyDepth = maxDepthSizePair.maxDepth - 5;

  /*
    Actually allocate the tree on chain
  */

  // define the address the tree will live at
  const treeKeypair = Keypair.generate();
```

```
// create and send the transaction to create the tree on ch
const tree = await createTree(connection, payer, treeKeypai

/*
  Create the actual NFT collection (using the normal Metapl
  (nothing special about compression here)
*/

// define the metadata to be used for creating the NFT coll
const collectionMetadataV3: CreateMetadataAccountArgsV3 = {
  data: {
    name: "Super Sweet NFT Collection",
    symbol: "SSNC",
    // specific json metadata for the collection
    uri: "https://supersweetcollection.notarealurl/collecti
    sellerFeeBasisPoints: 100,
    creators: [
      {
        address: payer.publicKey,
        verified: false,
        share: 100,
      },
    ],
    collection: null,
    uses: null,
  },
  isMutable: false,
  collectionDetails: null,
};

// create a full token mint and initialize the collection (
const collection = await createCollection(connection, payer

/*
  Mint a single compressed NFT
*/

const compressedNFTMetadata: MetadataArgs = {
  name: "NFT Name",
  symbol: collectionMetadataV3.data.symbol,
  // specific json metadata for each NFT
  uri: "https://supersweetcollection.notarealurl/token.json
  creators: [
    {
      address: payer.publicKey,
      verified: false,
      share: 100,
    },
    {
      address: testWallet.publicKey,
      verified: false,
      share: 0,
    },
  ],
  editionNonce: 0,
  uses: null,
  collection: null,
  primarySaleHappened: false,
  sellerFeeBasisPoints: 0,
  isMutable: false,
  // these values are taken from the Bubblegum package
  tokenProgramVersion: TokenProgramVersion.Original,
  tokenStandard: TokenStandard.NonFungible,
};

// fully mint a single compressed NFT to the payer
console.log(`Minting a single compressed NFT to ${payer.pub

await mintCompressedNFT(
  connection,
  payer,
  treeKeypair.publicKey,
  collection.mint,
  collection.metadataAccount,
  collection.masterEditionAccount,
  compressedNFTMetadata,
  // mint to this specific wallet (in this case, the tree o
  payer.publicKey,
);

// fully mint a single compressed NFT
console.log(`Minting a single compressed NFT to ${testWalle

await mintCompressedNFT(
  connection,
  payer,
  treeKeypair.publicKey,
  collection.mint,
  collection.metadataAccount,
  collection.masterEditionAccount,
  compressedNFTMetadata,
  // mint to this specific wallet (in this case, airdrop to
  testWallet.publicKey,
);

///////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////

// fetch the payer's final balance
balance = await connection.getBalance(payer.publicKey);

console.log(`===============================`);
console.log(
  "Total cost:",
  numberFormatter((initBalance - balance) / LAMPORTS_PER_SO
  "SOL\n",
);
})();
```

Hi there, I’m nitt, Ml engineer at Trustless engineering corp. I like to build ML models with large datasets. I would like to let you know that I got to know very late about this bounty and tried my best to curate the programs, but if you guys are finetuning an LLM on Solana code, I’d like to work with you guys. I’m experienced in curating datasets and finetuning models.

https://github.com/nithinexe
https://huggingface.co/nitt