

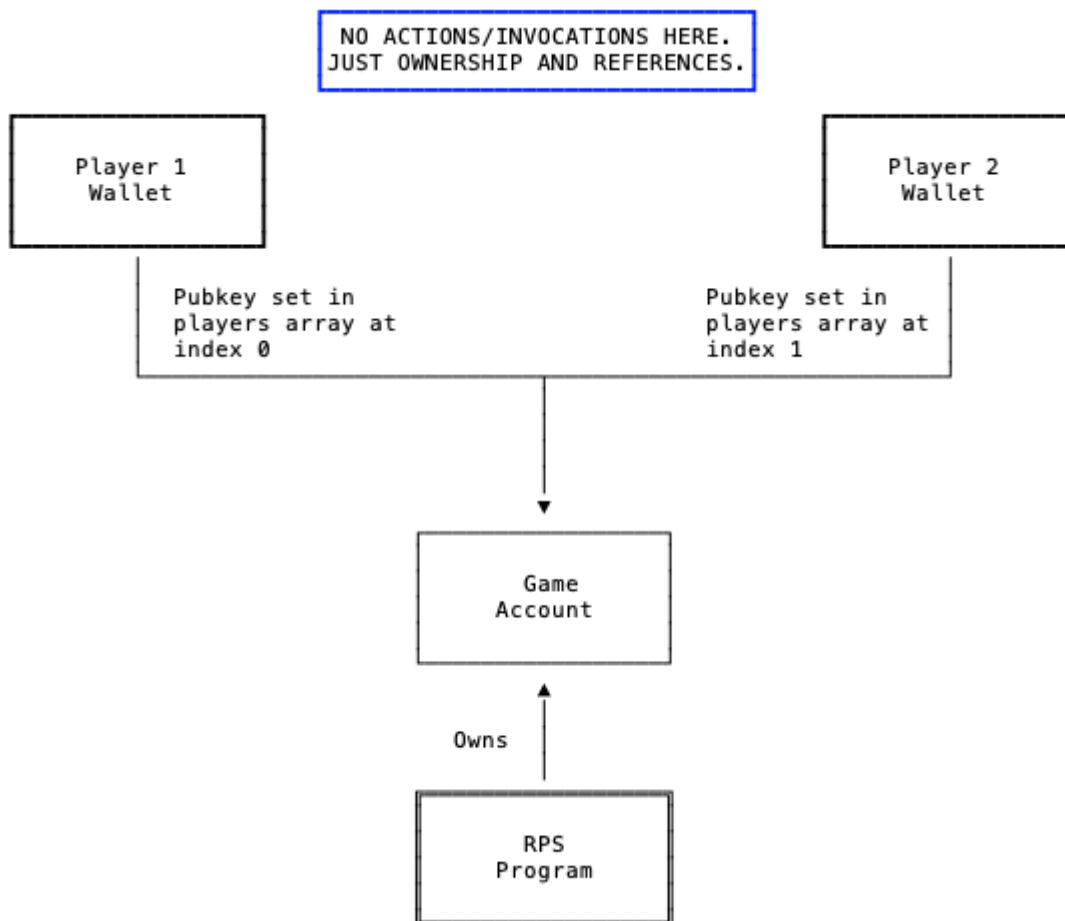
Lesson 15

Anchor examples

Rock Paper Scissors example

Program that implements rock-paper-scissors game.

Account diagram:



Flow:

1. Create game account: `new_game`

```
#[account]
pub struct Game {
    players: [Pubkey; 2],
    hashed_hand: [[u8; 32]; 2],
    hash_submitted: [bool; 2],
    hand: [Hand; 2],
    hand_submitted: [bool; 2],
    winner: String,
}
```

This account will store the state necessary for a single game such as:

- players public keys (array)
- hashes of hands (array)
- hands with salt (array)
- hands enum (array)
- flag for providing hash (array)
- winner

2. Player X submits hash containing their upcoming hand

3. Player Y submits hash containing their upcoming hand

4. Player X submits string containing their hand and the salt

5. Player Y submits string containing their hand and the salt

There is no required order for who submits hash first, but hands/salt strings can't be submitted until both hashes have been provided.

Versioned Transactions

See [Proposal](#)

Because of restrictions on the size of messages passed around the network, typically we cannot include more than 35 accounts or equivalent in a single transaction.

A Proposed solution is to

1. Allow address look up tables
2. Add a new transaction format to handle these.

After addresses are stored on-chain in an address lookup table account, they may be succinctly referenced in a transaction using a 1-byte u8 index rather than a full 32-byte address. This will require a new transaction format to make use of these succinct references as well as runtime handling for looking up and loading addresses from the on-chain lookup tables.

Address lookup tables must be rent-exempt when initialized and after each time new addresses are appended. Lookup tables can either be extended from an on-chain buffered list of addresses or directly by appending addresses through instruction data.

Once an address lookup table is no longer needed, it can be deactivated and closed to have its rent balance

reclaimed.

Versioned Transactions

In order to support address table lookups, the structure of serialized transactions must be modified. The new transaction format should not affect transaction processing in the Solana program runtime, invoked programs will be unaware of which transaction format was used.

The message header encodes `num_required_signatures` as a `u8`. Since the upper bit of the `u8` will never be set for a valid transaction, we can enable it to denote whether a transaction should be decoded with the versioned format or not.

Limitations

- Max of 256 unique accounts may be loaded by a transaction because `u8` is used by compiled instructions to index into transaction message `account_keys`.
- Address lookup tables can hold up to 256 entries because lookup table indexes are also `u8`.
- Transaction signers may not be loaded through an address lookup table, the full address of each signer must be serialized in the transaction. This

ensures that the performance of transaction signature checks is not affected.

- Hardware wallets will not be able to display details about accounts referenced through address lookup tables due to inability to verify on-chain data.
 - Only single level address lookup tables can be used. Recursive lookups will not be supported.
-

Solidity On Solana

See Solang [Article](#)

Hyperledger have introduced a Solidity compiler for Solana

It features

- Support for Solana programs such as SPL
- Anchor Development
- Support for Accounts and PDAs
- Compatibility with Solidity 0.8

Compiling a language is relatively straight forward, what is more difficult is integrating the Solana account model into Solidity.

This is mainly done via adding annotations to existing Solidity code.

Example Code

```
@program_id("F1ipperKF9EfD821ZbbYjS319LXYiBmj  
hzkkf5a26rC")  
contract starter {  
    bool private value = true;  
  
    @payer(payer)  
    constructor(address payer) {  
        print("Hello, World!");  
    }  
}
```

```

    }

    /// A message that can be called on
    instantiated contracts.

    /// This one flips the value of the
    stored `bool` from `true`
    /// to `false` and vice versa.
    function flip() public {
        value = !value;
    }

    /// Simply returns the current value of
    our `bool`.
    function get() public view returns (bool)
    {
        return value;
    }
}

```

Important differences

1. The `@program_id` annotation:

The `@program_id` annotation is used to specify the on-chain address of the program.

```

@program_id("F1ipperKF9EfD821ZbbYjS319LXYiBmj
hzkkf5a26rC") // on-chain program address

```

2. The `@payer` annotation:

When storing data on-chain, a certain amount of SOL needs to be allocated to cover the storage costs. The `@payer` annotation specifies the user that will pay the SOL required to create the account for storing the state variable.

```
@payer(payer) // payer for the "data account"  
constructor(address payer) {  
    print("Hello, World!");  
}
```

Storing data

- EVM smart contracts can directly store state variables.
- Solana on-chain programs, on the other hand, create separate accounts to hold state data. These are often referred to as "data accounts" and are "owned" by a program.

Installation

The first part of the installation is the same as installing Anchor, but I will detail the steps here anyway.

Requirements

- Rust
- Node.js
- WSL for Windows users

1. Install the [Solana Tool Suite](#)

```
sh -c "$(curl -sSfL  
https://release.solana.com/v1.17.14/install  
l)"
```

or

```
cmd /c "curl  
https://release.solana.com/v1.17.14/solana  
-install-init-x86_64-pc-windows-msvc.exe -  
-output C:\solana-install-tmp\solana-  
install-init.exe --create-dirs" on Windows
```

2. Install Anchor with

```
cargo install --git  
https://github.com/coral-xyz/anchor  
anchor-cli --locked --force
```

Creating a new project

```
anchor init project_name --solidity
```

The example solidity code you get is

```
@program_id("F1ipperKF9EfD821ZbbYjS319LXYiBmj  
hzkkf5a26rC")  
contract my_prog {  
    bool private value = true;
```

```

    @payer(payer)
    constructor(address payer) {
        print("Hello, World!");
    }

    /// A message that can be called on
    instantiated contracts.
    /// This one flips the value of the
    stored `bool` from `true`
    /// to `false` and vice versa.
    function flip() public {
        value = !value;
    }

    /// Simply returns the current value of
    our `bool`.
    function get() public view returns (bool)
    {
        return value;
    }
}

```

More insight is gained by looking at typescript code used to test this

```

import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { my_prog } from

```

```
 "../target/types/my_prog";

describe("my_prog", () => {

// Configure the client to use the local
cluster.
const provider = anchor.AnchorProvider.env();
anchor.setProvider(provider);

const dataAccount =
anchor.web3.Keypair.generate();
const wallet = provider.wallet;
const program = anchor.workspace.my_prog as
Program<my_prog>;

it("Is initialized!", async () => {

// Add your test here.

const tx = await
program.methods.new(wallet.publicKey)
.accounts({ dataAccount:
dataAccount.publicKey })
.signers([dataAccount]).rpc();

console.log("Your transaction signature",
tx);

const val1 = await program.methods.get()
.accounts({ dataAccount:
```

```
dataAccount.publicKey })
.view();

console.log("state", val1);
await program.methods.flip()
.accounts({ dataAccount:
dataAccount.publicKey })
.rpc();

const val2 = await program.methods.get()
.accounts({ dataAccount:
dataAccount.publicKey })
.view();

console.log("state", val2); });

});
```

Example Token Creation

```
import "../libraries/spl_token.sol";
import "../libraries/mpl_metadata.sol";

@program_id("8eZPhSaXfHqbcrrfskVThtCG68kq8MfVH
```

```
qmtm6wYf4TLb")
```

```
contract create_token {
```

```
    // Creating a dataAccount is required by  
    Solang
```

```
    // The account is unused in this example
```

```
    @payer(payer) // payer account
```

```
    constructor() {}
```

```
    function createTokenMint(
```

```
        address payer, // payer account
```

```
        address mint, // mint account to be  
        created
```

```
        address mintAuthority, // mint  
        authority for the mint account
```

```
        address freezeAuthority, // freeze  
        authority for the mint account
```

```
        address metadata, // metadata account  
        to be created
```

```
        uint8 decimals, // decimals for the  
        mint account
```

```
        string name, // name for the metadata  
        account
```

```
        string symbol, // symbol for the  
        metadata account
```

```
        string uri // uri for the metadata  
        account
```

```
    ) public {
```

```
        // Invoke System Program to create a  
        new account for the mint account and,
```

```

        // Invoke Token Program to initialize
the mint account
        // Set mint authority, freeze
authority, and decimals for the mint account
        SplToken.create_mint(
            payer,                // payer
account
            mint,                // mint account
            mintAuthority,       // mint
authority
            freezeAuthority,     // freeze
authority
            decimals              // decimals
        );

        // Invoke Metadata Program to create
a new account for the metadata account
        MplMetadata.create_metadata_account(
            metadata, // metadata account
            mint,     // mint account
            mintAuthority, // mint authority
            payer,    // payer
            payer,    // update authority (of
the metadata account)
            name,     // name
            symbol,   // symbol
            uri // uri (off-chain metadata
json)
        );

```

}

}

The token library is [here](#)

Other Languages

Seahorse

[Docs](#)

Seahorse lets you write Solana programs in Python. It is a community-led project built on [Anchor](#).

Install with

```
cargo install seahorse-lang
```

It is also available in the Solana playground.