Lesson 13

This Week

Anchor continued
Solana Program Library
Security
NFTs / Metaplex

> ## Anchor Summary
>
> An Anchor program consists of three parts.
>
> 1. The `program` module,
> 2. the Accounts structs which are marked with `#[derive(Accounts)]`, and
> 3. the `declare_id` macro.
>
> The `program` module is where you write your business logic.
> The Accounts structs is where you validate accounts.
> The `declare_id` macro creates an `ID` field that stores the address of your program.
> Anchor uses this hardcoded `ID` for security checks and it also allows other crates to access your program's address.
>
> For example a boilerplate Anchor program would look like

```
// use this import to gain access to common anchor features
use anchor_lang::prelude::*;


// declare an id for your program
declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");


// write your business logic here
#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(_ctx: Context<Initialize>) -> Result<()> {
        Ok(())
    }
}


// validate incoming accounts here
```

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

Anchor - the program module

The program module is where we define our business logic

We have seen this in the lottery example

```
  #[program]

mod example1 {

use super::*;

// Creates an account for the lottery

pub fn initialise_lottery(ctx: Context<Create>, ticket_price: u64, oracle_pubkey:
Pubkey) -> Result<()> {

let lottery: &mut Account<Lottery> = &mut ctx.accounts.lottery;

lottery.authority = ctx.accounts.admin.key();

lottery.count = 0;

lottery.ticket_price = ticket_price;

lottery.oracle = oracle_pubkey;

Ok(())

}
```

The above function initialises our lottery, using the Create Context

```
  #[derive(Accounts)]

pub struct Create<'info> {

#[account(init, payer = admin, space = 8 + 180)]

pub lottery: Account<'info, Lottery>,

#[account(mut)]

pub admin: Signer<'info>,

pub system_program: Program<'info, System>,
```

```
}
```

Any function that is part of the API of the program takes a `Context` type as its first argument. Through this context argument it can access the accounts (`ctx.accounts`), the program id (`ctx.program_id`) of the executing program,
and the remaining accounts (`ctx.remaining_accounts`).
`remaining_accounts` is a vector that contains all accounts that were passed into the instruction but are not declared in the `Accounts` struct.

Instruction Data

If your function needs instruction data, you can add arguments after the context.
Anchor will then automatically deserialise the instruction data into the arguments.

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn set_data(ctx: Context<SetData>, data: Data) -> Result<()> {
        ctx.accounts.my_account.data = data.data;
        ctx.accounts.my_account.age = data.age;
        Ok(())
    }
}

#[account]
#[derive(Default)]
pub struct MyAccount {
    pub data: u64,
    pub age: u8
}

#[derive(AnchorSerialize, AnchorDeserialize, Eq, PartialEq, Clone, Copy, Debug)]
pub struct Data {
    pub data: u64,
    pub age: u8
}
```

Instruction Data

If your function needs instruction data, you can add arguments after the context. Anchor will then automatically deserialise the instruction data into the arguments.

```
#[program]
mod hello_anchor {
    use super::*;
    pub fn set_data(ctx: Context<SetData>, data: Data) -> Result<()> {
        ctx.accounts.my_account.data = data.data;
        ctx.accounts.my_account.age = data.age;
        Ok(())
    }
}

#[account]
#[derive(Default)]
pub struct MyAccount {
    pub data: u64,
    pub age: u8
}

#[derive(AnchorSerialize, AnchorDeserialize, Eq, PartialEq, Clone, Copy, Debug)]
pub struct Data {
    pub data: u64,
    pub age: u8
}
```

SPL

Solana Program Library covers

- tokens
- governance
- name service
- token swaps
- lending

The Solana token program is heavily used, its program id is
: TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA

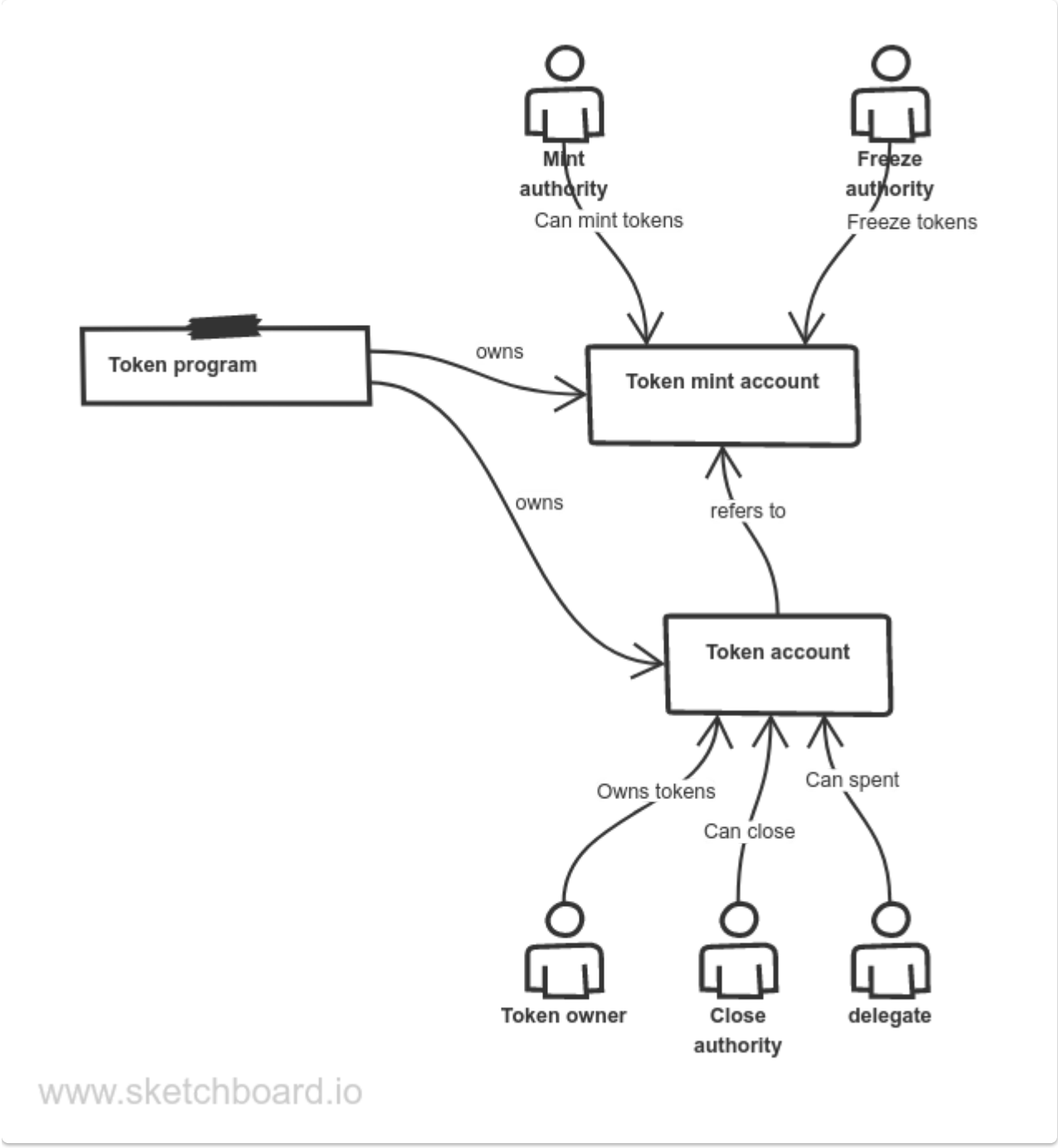
Documentation can be found at docs

Token Program

Creating a new token type

A new token type can be created by initialising a new Mint with the `InitializeMint` instruction. The Mint is used to create or "mint" new tokens, and these tokens are stored in Accounts. A Mint is associated with each Account, which means that the total supply of a particular token type is equal to the balances of all the associated Accounts.

The process of creating a new token type



We first call `InitializeMint`
This takes some parameters that are stored in a struct

```language-none
pub struct Mint {
/// Optional authority used to mint new tokens. The mint authority may only be provided during mint creation. If no mint authority is present then the mint has a fixed supply and no further tokens may be minted.
pub mint_authority: COption<Pubkey>,

/// Total supply of tokens.
pub supply: u64,

/// Number of base 10 digits to the right of the decimal place.
pub decimals: u8,

/// Is `true` if this structure has been initialized
pub is_initialized: bool,

/// Optional authority to freeze token accounts.
```

```
pub freeze_authority: COption<Pubkey>,

}
```

Creating accounts

Accounts hold token balances and are created using the `InitializeAccount` instruction. Each Account has an owner who must be present as a signer in some instructions.

An Account's owner may transfer ownership of an account to another using the `SetAuthority` instruction.

It's important to note that the `InitializeAccount` instruction does not require the Solana account being initialised also be a signer. The `InitializeAccount` instruction should be atomically processed with the system instruction that creates the Solana account by including both instructions in the same transaction.

```language-none
pub struct Account {
/// The mint associated with this account
pub mint: Pubkey,
/// The owner of this account.
pub owner: Pubkey,
/// The amount of tokens this account holds.
pub amount: u64,
/// If `delegate` is `Some` then `delegated_amount` represents
/// the amount authorized by the delegate
pub delegate: COption<Pubkey>,
/// The account's state
pub state: AccountState,
/// If is_native.is_some, this is a native token, and the value logs the rent-
exempt reserve. An
/// Account is required to be rent-exempt, so the value is used by the
Processor to ensure that
/// wrapped SOL accounts do not drop below this threshold.
pub is_native: COption<u64>,
/// The amount delegated
pub delegated_amount: u64,
/// Optional authority to close the account.
pub close_authority: COption<Pubkey>,

}
```

Next the `MintTo` instruction is called, taking

- Public key of the mint
- Address of the token account to mint to
- The mint authority
- Amount to mint
- Signing accounts if `authority` is a multisig
- SPL Token program account

This will mint tokens to the destination account

Transfer

To transfer tokens we invoke the function `process_transfer` this transfers a certain amount of token from a source account to a destination account:
We pass in the source and destination accounts and the amount.
The program will check that

1. Neither source account nor destination account is frozen
2. The source account's mint and destination account's mint are the same
3. The transferred `amount` is no more than source account's token amount

The owner of the source Account must be present as a signer in the `Transfer` instruction when the source and destination accounts are different.

Note the source and destination can be the same, if so the `Transfer` will *always* succeed.
Therefore, a successful `Transfer` does not necessarily imply that the involved Accounts were valid SPL Token accounts, that any tokens were moved, or that the source Account was present as a signer.
It is recommended to check that the source and destination are different before calling the transfer function.

BURN

Burn is the opposite of Mint and removes tokens, from the supply and the given account.

Approve

This allows transfer of a certain amount by a delegate.

- Only one delegate is possible per account / token.
- A new approval will overrride the previous one.

Revoke

Removes the approval

Freeze / Thaw Account

This will freeze / unfreeze the account preventing / allowing transfers / mints to it.

Associated Token Program

From Solana Docs
A user may own arbitrarily many token accounts belonging to the same mint which makes it difficult for other users to know which account they should send tokens to and introduces friction into many other aspects of token management.
The associated token program introduces a way to deterministically derive a token account key from a user's main System account address and a token mint address, allowing the user to create a main token account for each token they own.

We call these accounts *Associated Token Accounts*.

In addition, it allows a user to send tokens to another user even if the beneficiary does not yet have a token account for that mint. Unlike a system transfer, for a token transfer to succeed the recipient must have a token account with the compatible mint already, and somebody needs to fund that token account. If the recipient must fund it first, it makes things like airdrop campaigns difficult and just generally increases the friction of token transfers.
The associated token program allows the sender to create the associated token account for the receiver, so the token transfer just works.

The associated token account for a given wallet address is simply a program-derived account from the wallet address itself and the token mint.
This gives us a way to deterministically find an account address based on the wallet and the mint account.

Finding the Associated token addresses

The get_associated_token_address Rust function may be used by clients to derive the wallet's associated token address.

The associated account address can be derived in TypeScript with:

```
import { PublicKey } from '@solana/web3.js';                    language-none
import { TOKEN_PROGRAM_ID } from '@solana/spl-token';

const SPL_ASSOCIATED_TOKEN_ACCOUNT_PROGRAM_ID: PublicKey = new PublicKey(
  'ATokenGPvbdGVxr1b2hvZbsiqW5xWH25efTNsLJA8knL',
);

async function findAssociatedTokenAddress(
    walletAddress: PublicKey,
    tokenMintAddress: PublicKey
): Promise<PublicKey> {
    return (await PublicKey.findProgramAddress(
        [
            walletAddress.toBuffer(),
            TOKEN_PROGRAM_ID.toBuffer(),
            tokenMintAddress.toBuffer(),
        ],
        SPL_ASSOCIATED_TOKEN_ACCOUNT_PROGRAM_ID
    ))[0];
}
```

Creating the Associated Token Account

If the associated token account for a given wallet address does not yet exist, it may be created by *anybody* by issuing a transaction containing the instruction returned by create_associated_token_account.

Regardless of creator the new associated token account will be fully owned by the wallet, as if the wallet itself had created it.

This article looks at tokens and accounts in depth, and has some good diagrams to show the relationship between the different accounts.

Token Swap Program

AMM Background

A pool is set up to provide liquidity for a token pair

User Process

The user can interact with the token swap program by

1. Providing Liquidity
2. Swapping one token for the other.

Providing Liquidity

A pool needs to be created, then once it exists, users can add liquidity
When a user adds liquidity to a pool (lets say a pool of tokens A and B), by supplying both tokens, they will receive a share of the pool in the form of LP tokens.
These LP tokens can later be redeemed to get back tokens A and B.
To incentivise the provision of liquidity, the user will receive more tokens than they originally supplied, the source of this extra amount comes from fees that users pay when they swap tokens.

Creating a new token swap pooL

Imagine we wish to create a pool for two tokens "A" and "B".
For this we need the following accounts

- empty pool state account
  The pool state account simply needs to be created
  using `system_instruction::create_account` with the correct size and enough lamports to be rent-free.
- pool authority
  The pool authority is a [program derived address]that can "sign" instructions towards other programs.
  This is required for the Token Swap Program to mint pool tokens and transfer tokens from its token A and B accounts.
- token A account
- token B account
- pool token mint
  The token A / B accounts, pool token mint, and pool token accounts must all be created (using `system_instruction::create_account`) and initialised
  (using `spl_token::instruction::initialize_mint` or `spl_token::instruction::initialize_account`).
  The token A and B accounts must be funded with tokens, and their owner set to the swap authority, and the mint must also be owned by the swap authority.
- pool token fee account
- pool token recipient account
- token program

Once all of these accounts are created, the Token Swap `initialize` instruction will properly set everything up and allow for immediate trading. Note that the pool state account is not required to be a signer on `initialize`, so it's important to perform the `initialize` instruction in the same transaction as its `system_instruction::create_account`.

Depositing Liquidity

Use the `deposit_all_token_types` or `deposit_single_token_type_exact_amount_in` instructions to add liquidity to the pool in exchange for LP tokens.

The user will need to approve a delegate to transfer tokens from their own A and B token accounts. This limits the amount of tokens that can be taken from the user's account by the program.

Swapping tokens

Once a pool is created, users can immediately begin trading on it using the `swap` instruction. The swap instruction transfers tokens from a user's source account into the swap's source token account, and then transfers tokens from its destination token account into the user's destination token account.

Since Solana programs require all accounts to be declared in the instruction, users need to gather all account information from the pool state account: the token A and B accounts, pool token mint, and fee account.

Additionally, the user must allow for tokens to be transferred from their source token account. The best practice is to `spl_token::instruction::approve` a precise amount to a new throwaway Keypair, and then have that new Keypair sign the swap transaction. This limits the amount of tokens that can be taken from the user's account by the program.

You can see the order of operations in the test files in SPL

```language-none
async function main() {

// These test cases are designed to run sequentially and in the following
order

console.log('Run test: createTokenSwap (constant price)');

await createTokenSwap(CurveType.ConstantPrice, new Numberu64(1));

console.log(

'Run test: createTokenSwap (constant product, used further in tests)',

);

await createTokenSwap(CurveType.ConstantProduct);

console.log('Run test: deposit all token types');

await depositAllTokenTypes();

console.log('Run test: withdraw all token types');

await withdrawAllTokenTypes();

console.log('Run test: swap');

await swap();

console.log('Run test: create account, approve, swap all at once');

await createAccountAndSwapAtomic();
```

```
console.log('Run test: deposit one exact amount in');

await depositSingleTokenTypeExactAmountIn();

console.log('Run test: withrdaw one exact amount out');

await withdrawSingleTokenTypeExactAmountOut();

console.log('Success\n');

}
```

The swap test

```language-none
export async function swap(): Promise<void> {

console.log('Creating swap token a account');

let userAccountA = await mintA.createAccount(owner.publicKey);

await mintA.mintTo(userAccountA, owner, [], SWAP_AMOUNT_IN);

const userTransferAuthority = new Account();

await mintA.approve(

userAccountA,

userTransferAuthority.publicKey,

owner,

[],

SWAP_AMOUNT_IN,

);

console.log('Creating swap token b account');

let userAccountB = await mintB.createAccount(owner.publicKey);

let poolAccount = SWAP_PROGRAM_OWNER_FEE_ADDRESS

? await tokenPool.createAccount(owner.publicKey)

: null;

const confirmOptions = {

skipPreflight: true

}

console.log('Swapping');

await tokenSwap.swap(

userAccountA,

tokenAccountA,

tokenAccountB,
```

```
userAccountB,

poolAccount,

userTransferAuthority,

SWAP_AMOUNT_IN,

SWAP_AMOUNT_OUT,

confirmOptions

);
```

Blockchain Governance

“The greatest challenge that new blockchains must solve isn’t speed or scaling, it’s governance”
Kai Sedgwick - Why Governance is the Greatest Problem for Blockchains To Solve**

It is useful to think of governance in the following areas

- Consensus
  Who is involved and how do they come to consensus ?
- Information
  How does relevant information reach the participants ?
- Incentives
  How are the incentives aligned to ensure
- Correct Behaviour
  There is a sufficient level of participation
- Procedures
  In a decentralised system how are
  Proposals made
  Votes submitted
  Consensus reached

Types of Governance

1. Off chain
   The mechanism to change the protocol are external to the system
   The process is often
   > - ad hoc
   > - may be poorly specified
   > - communication and coordination can be problematic
   >   Developers may have a key role in deciding and implementing changes to the protocol
2. On chain
   The mechanism to change the protocol is part of the protocol
   Typically participants can vote to accept or reject proposals to upgrade the protocol or some aspects of the system
   Coordination and communication is usually more efficient than in off chain solutions

In reality, there is often a mixture of both

Governance in Solana

See Docs

The Feature Proposal Program provides a workflow for activation of Solana network features through community vote based on validator stake weight.

Community voting is accomplished using SPL Tokens. Tokens are minted that represent the total active stake on the network, and distributed to all validators based on their stake. Validators vote

for feature activation by transferring their vote tokens to a predetermined address. Once the vote threshold is met the feature is activated.
The Feature Proposal Program provides an additional mechanism over these runtime feature activation primitives to permit feature activation by community vote when appropriate.

Lifecycle

1. Implement the feature
   The developers change the runtime to include a possible new feature
2. Initiate the voting
3. Tally the Votes - if the vote succedes the feature is implemented.

Feature Proposal Program

There is a CLI
`cargo install spl-feature-proposal-cli`

Lifecycle

Implement the Feature

The first step is to conceive of the new feature and realize it in the Solana code base, working with the core Solana developers at https://github.com/solana-labs/solana.

During the implementation, a *feature id* will be required to identity the new feature in the code base to avoid the new functionality until its activation.

Initiate the Feature Proposal

After the feature is implemented and deployed to the Solana cluster, the *feature id* will be visible in `solana feature status` and the *feature proposer* may initiate the community proposal process.

COST: As a part of token distribution, the *feature proposer* will be financing the creation of SPL Token accounts for each of the validators. A SPL Token account requires 0.00203928 SOL at creation, so the cost for initiating a feature proposal on a network with 500 validators is approximately 1 SOL.

After advertising to the validators that a feature proposal is pending their acceptance, the votes are tallied by running:

```
$ spl-feature-proposal tally 8CyUVvio2oYAP28ZkMBPHq88ikhRgWet6i4NYsCW5Cxa
```

Anybody may tally the vote. Once the required number of votes are tallied, the feature will be automatically activated at the start of the next epoch.

Upon a successful activation the feature will now show as activated by `solana feature status` as well.

SPL Governance

See documentation

It provides building blocks that can be used with

- DAOs
- Authority providers for access control
- Multisig control for a wallet or upgrade authority

It has a modular construction and allows customisation via external plugins, which can be ordinary Solana (Anchor) programs.

DAO introduction

A decentralised autonomous organisation (DAO) is a community with a shared bank account. Members of the DAO make decisions in a transparent and decentralised fashion, with smart contracts executing these decisions.

As a result, the DAO structure provides a “flat” organisational structure.
Each DAO member has a voice in the community and the opportunity to drive the direction of the organisation.

Creating a multisig DAO

There is a frontend at Realms

You need to add

1. The name of your DAO;
2. The approval quorum, that is the minimum amount of yes votes to accept a proposal; and
3. People who'll be part of your team, whose will own a council token.

Create a bespoke DAO

This gives you more options, such as council settings and associated tokens, the governance program to use etc.

Create NFT Community DAO

Here NFTs are used to authorise voting.
This can be further expanded with different NFT collections having different meaning.

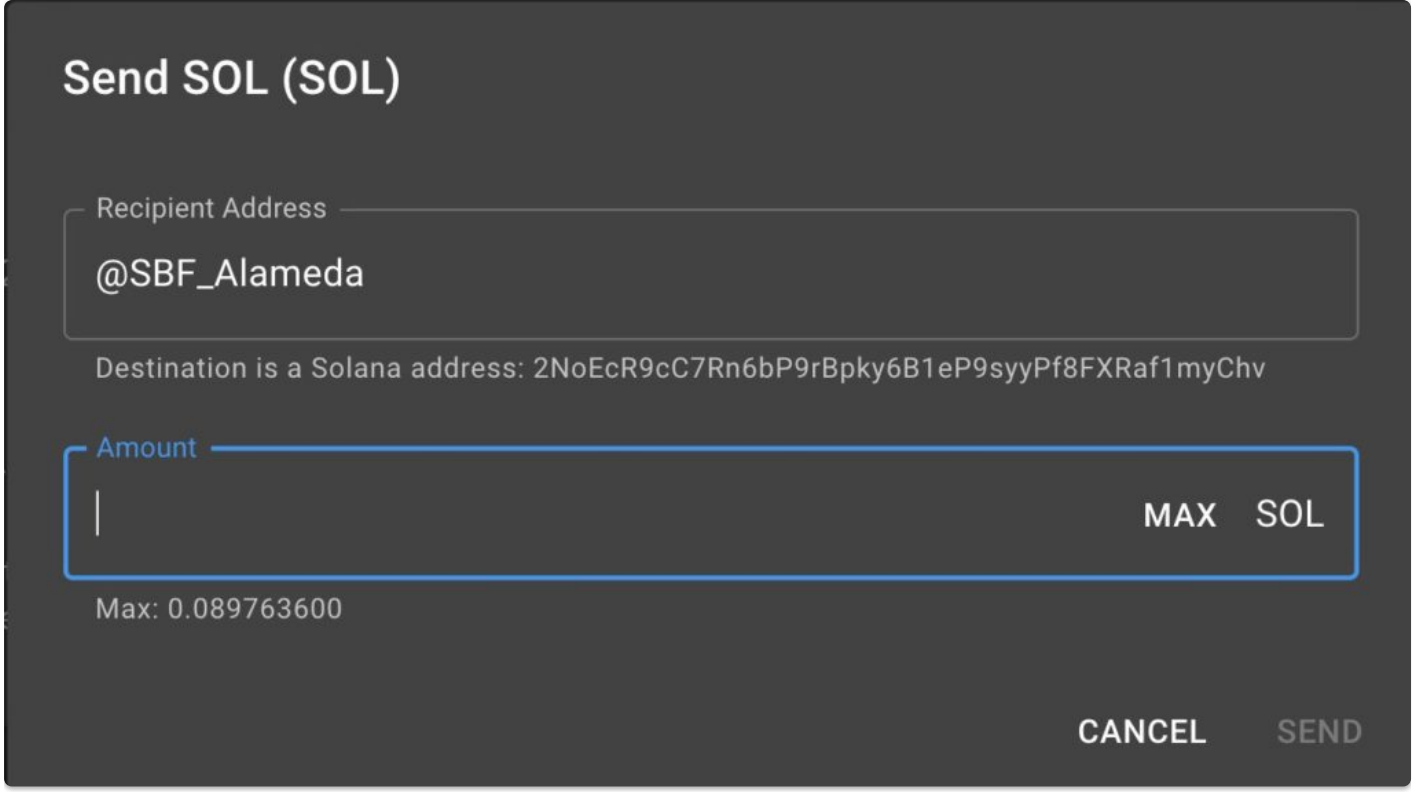
Name Service Program

This is Solana's implementation of Blockchain Naming Service (BNS). Name service manages ownership of a link to a given content such as IPFS CID or a twitter handle to a given Solana on-chain public key. Authority for modifcations of these mappings rests solely with the owner.

Naming services can be used for as pointers to:

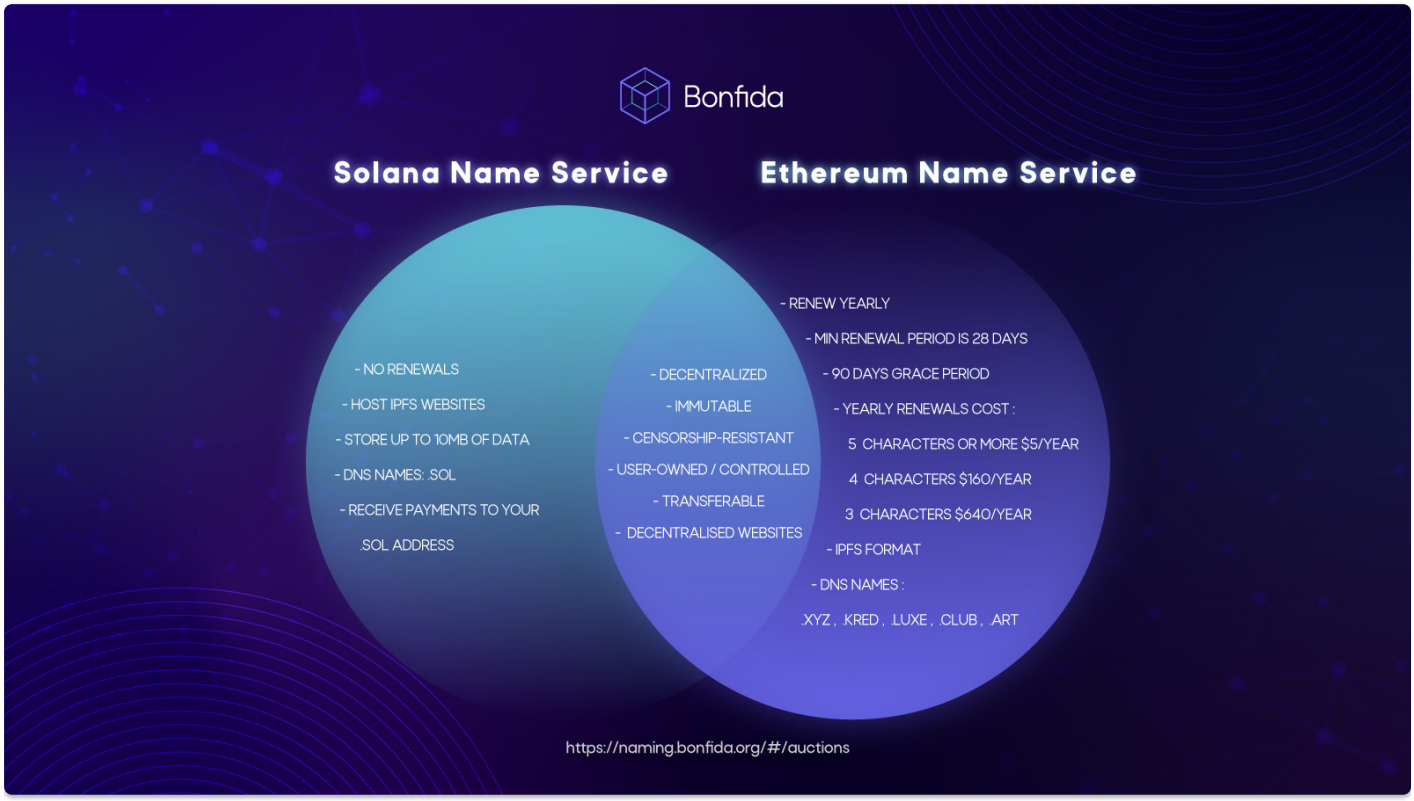
- music
- blogs
- online stores
- events

Human readable mappings can simplify and reduce rate of errors for transfers, as only the handle rather then the full key would need to be remembered.


Send SOL (SOL)
Recipient Address
@SBF_Alameda
Destination is a Solana address: 2NoEcR9cC7Rn6bP9rBpky6B1eP9syyPf8FXRaf1myChv
Amount
MAX
SOL
Max: 0.089763600
CANCEL
SEND

Example Bonfida

Bonfida is an on-chain DAO with a user interface built on the Serum dex.



Solana-pay

What is solana pay

Solana Pay is a standardised protocol that can be hosted on any server and retrieved with ease by the client-side application used by buyers and sellers alike. The response includes all the accounts and parameters necessary to execute a given set of transactions.

It can be used by developers to use it as a building block for further expansions and features such as:

- taxation management
- receipt generation
- graphical inventory display
- integration with till or existing POS hardware
- loyalties

Many protocols have been built on top of it, such as winners from the recent Riptide hackathon:

- mntPAY
- Phoria
- Radiant Pay
- Comerce DAO
- YamiPay

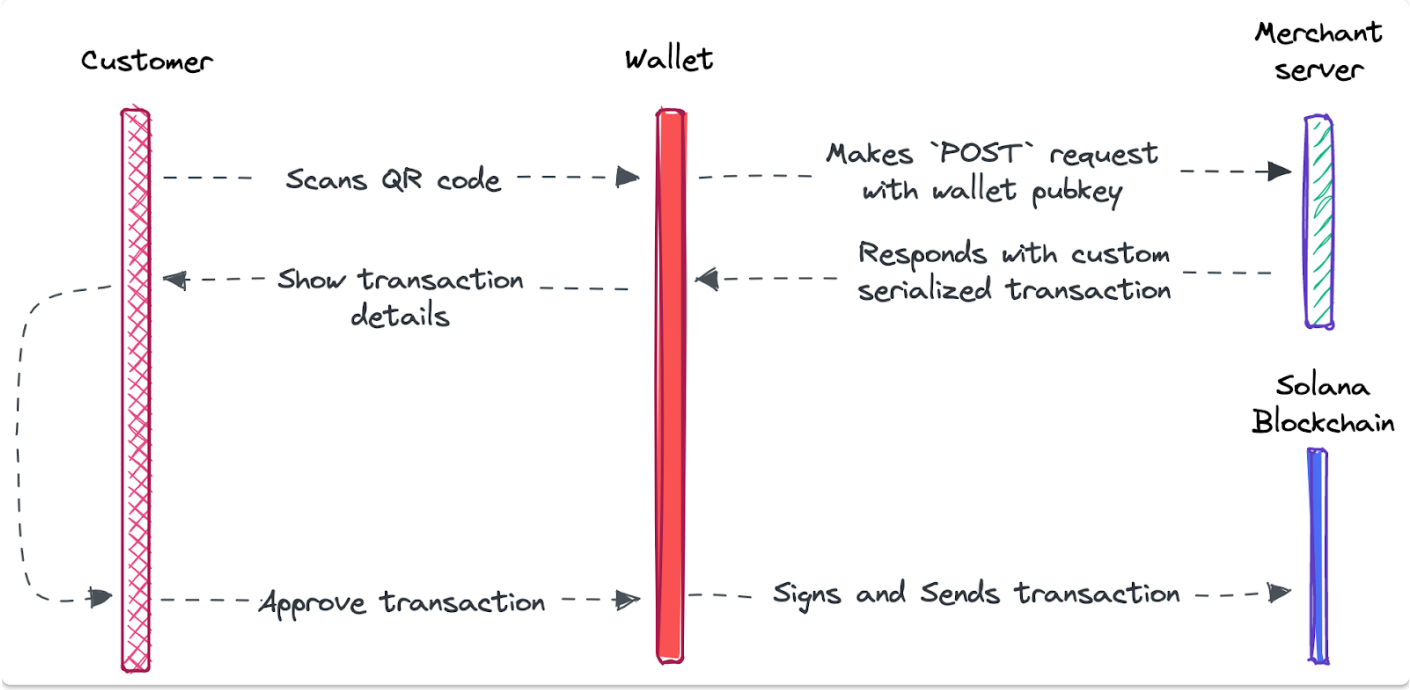
In essence, Solana-Pay can be thought of as being the same way as Serum being a foundation or a building block for other protocols such as Raydium or Mango Market. Each of those projects extends Solana-Pay by solving existing shortcomings or by improving customer experience.

Alternative services

Within Web 3.0 space BTCPay (Lightning Network) can be thought of as payment platform focused on consumer in person purchases.
In the traditional finance world similar functionality is provided by likes of Visa or Paypal.

How does it work

1. The customer scans a merchant QR code, which their wallet app interprets as a Solana Pay transaction request URL.
2. The wallet makes an HTTP request to the merchant API.
3. The merchant receives the wallet address in the request and can respond with a customised transaction for the customer.
4. The wallet shows the transaction details to the customer just like any other transaction, and can also display a merchant URL and icon.
5. The customer approves (or declines) the transaction, signing with their private key, and sending the transaction to the network.



Transaction template looks as follows:

```
solana:<recipient>
      ?amount=<amount>
      &spl-token=<spl-token>
      &reference=<reference>
      &label=<label>
      &message=<message>
      &memo=<memo>
```

Support

Solana-Pay supports currently the following mobile wallets:

- Phantom
- Glow
- Slope
- Crypto Please
- Solflare
- FTX

Payment tutorial

Device and account set-up

For this to work two wallets are required:

- Merchant, on the point of sale terminal
- Buyer, on the phone

Both phone and computer need to be set to devnet

Ensure Phantom wallet is installed on the phone, and the associated account has a SOL balance.

This can be achieved with:

```
solana airdrop <AMOUN_BELOW_2> <RECIPIENT_ADDRESS>
```

Solana-pay demo installation

1. Clone the repo

```
git clone https://github.com/solana-labs/solana-pay.git
```

2. Install dependencies

```
cd solana-pay/point-of-sale
yarn install
```

If it fails to install for whatever reason run:

```
npm i
```

or manually add missing libraries with:

```
yard add <package>
```

or

```
npm i <package>
```

3. Start a local instance of the solana-pay server

In a terminal run:

```
yarn dev
```

4. Start merchant proxy

Open a new terminal and run:

```
yarn proxy
```

5. Create a merchant address

https://localhost:3001?recipient=Your+Merchant+Address&label=Your+Store+Name

Use pubkey of your Phantom wallet and any name.

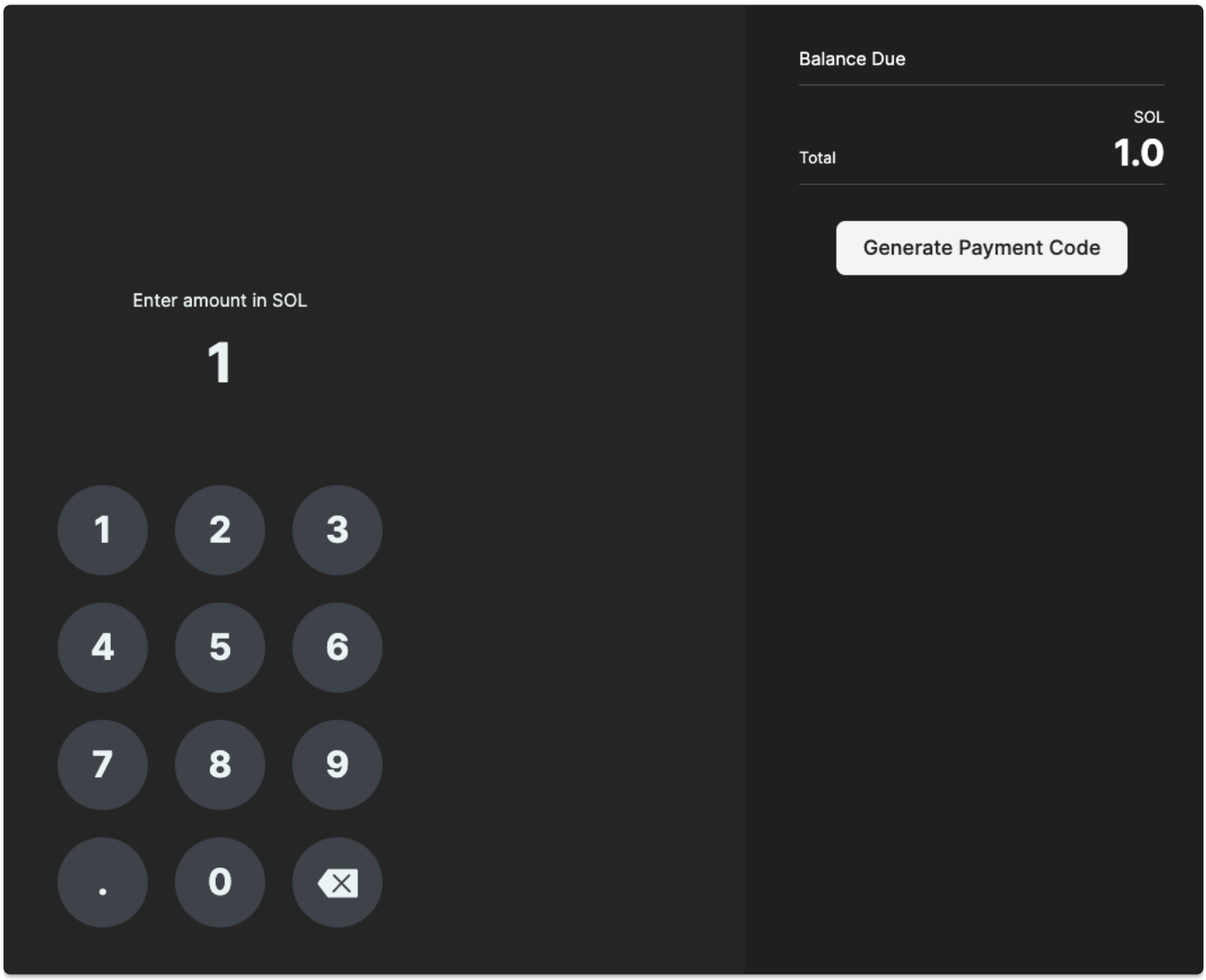
For example:

https://localhost:3001?
recipient=D3SaEUyEJporUcSmmLSiqyi7TuwEY8xnfP5jAAokVXdt&label=Extropy

6. Paste merchant address to your browser

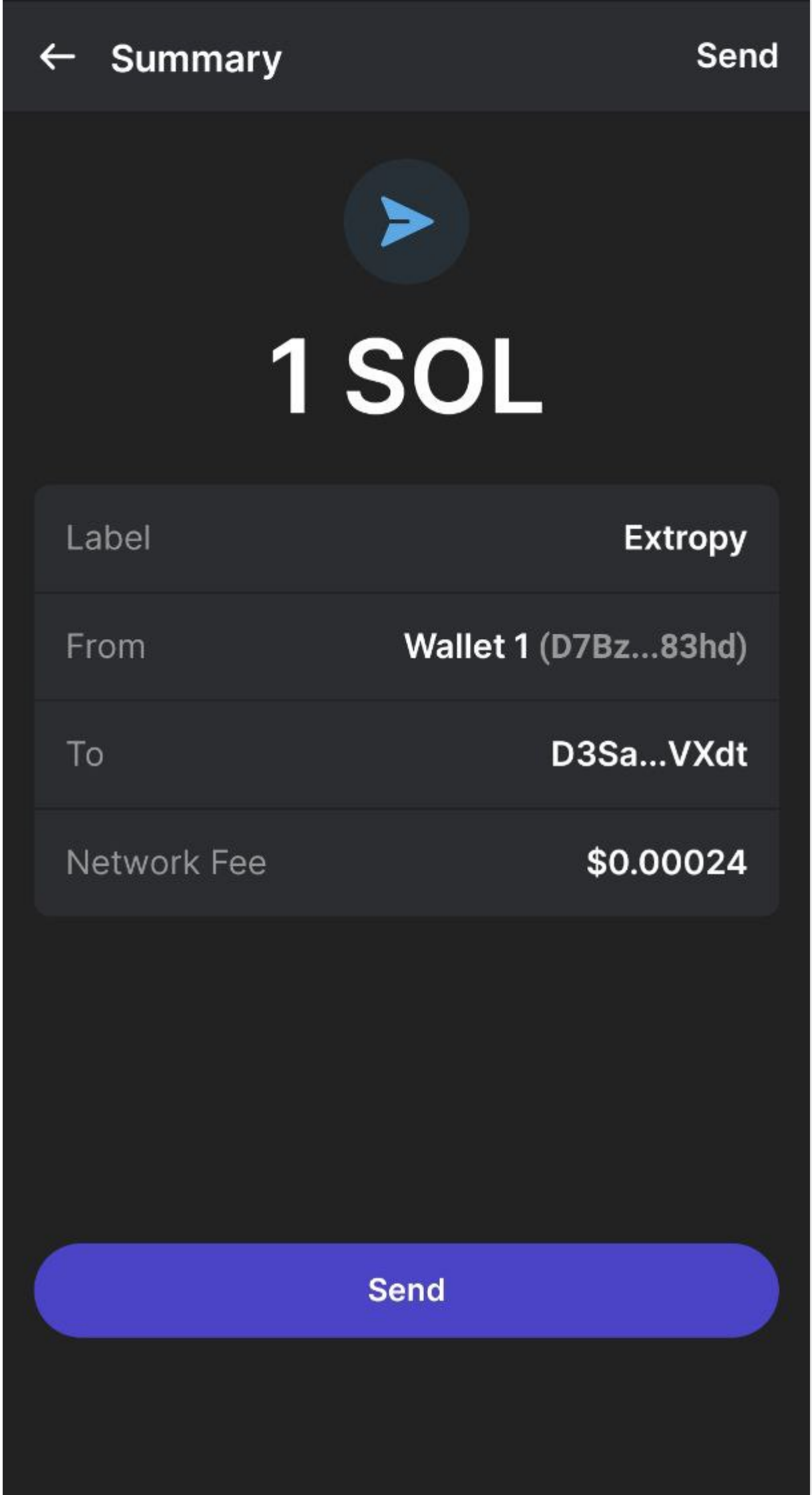
Payment

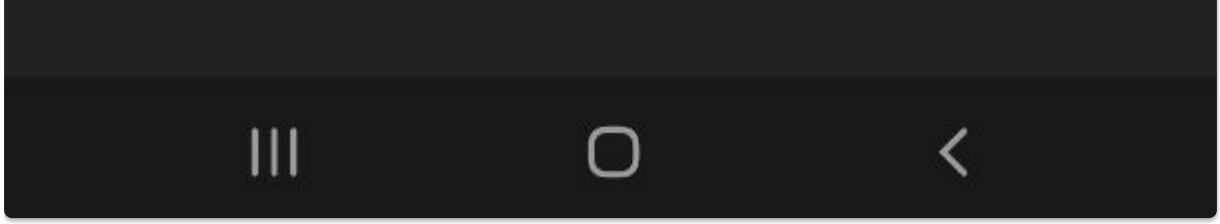
1. On the merchant POS application type out the amount to pay



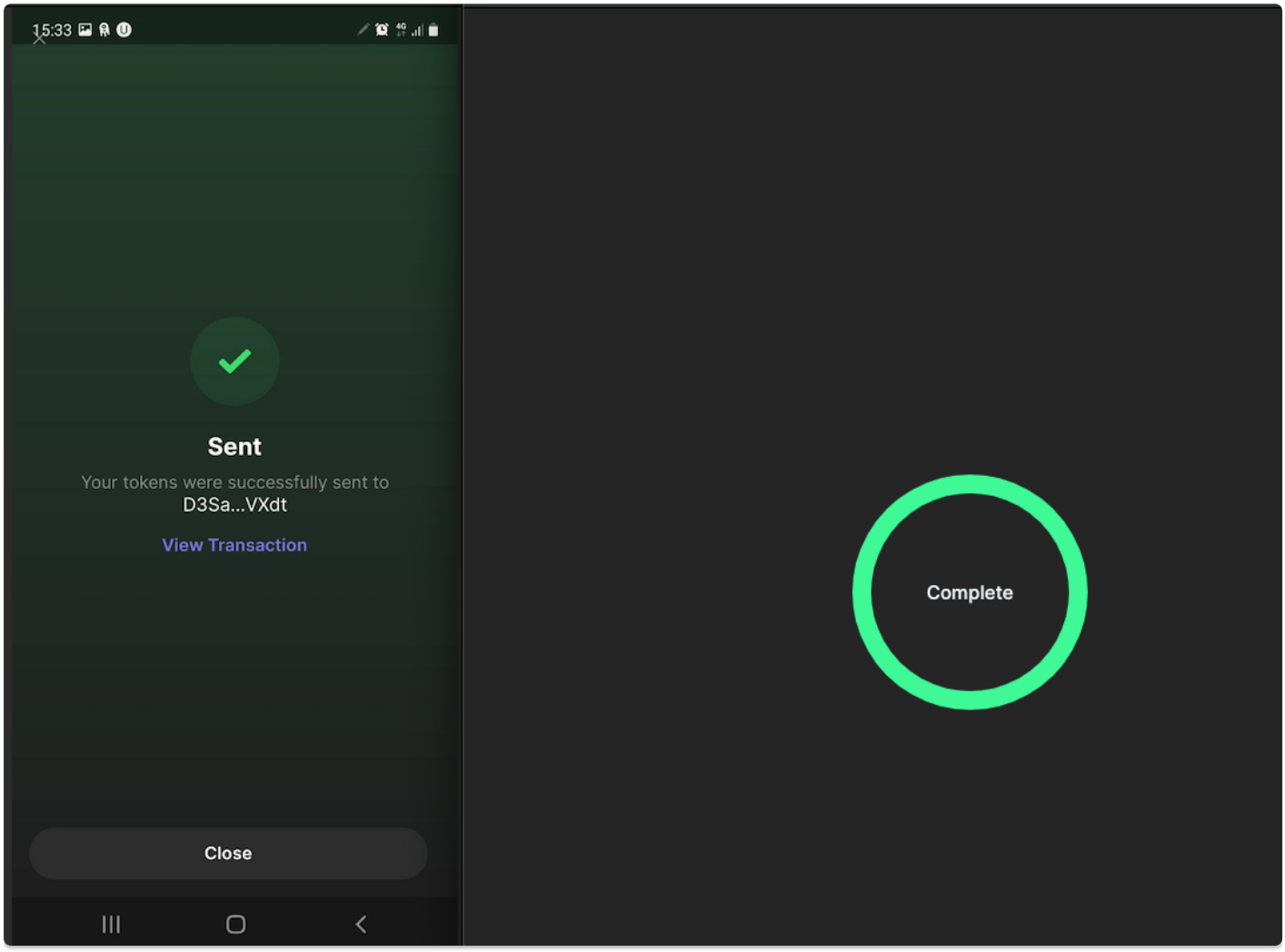
2. Scan the QR code with the camera

3. Confirm transaction via Phantom wallet

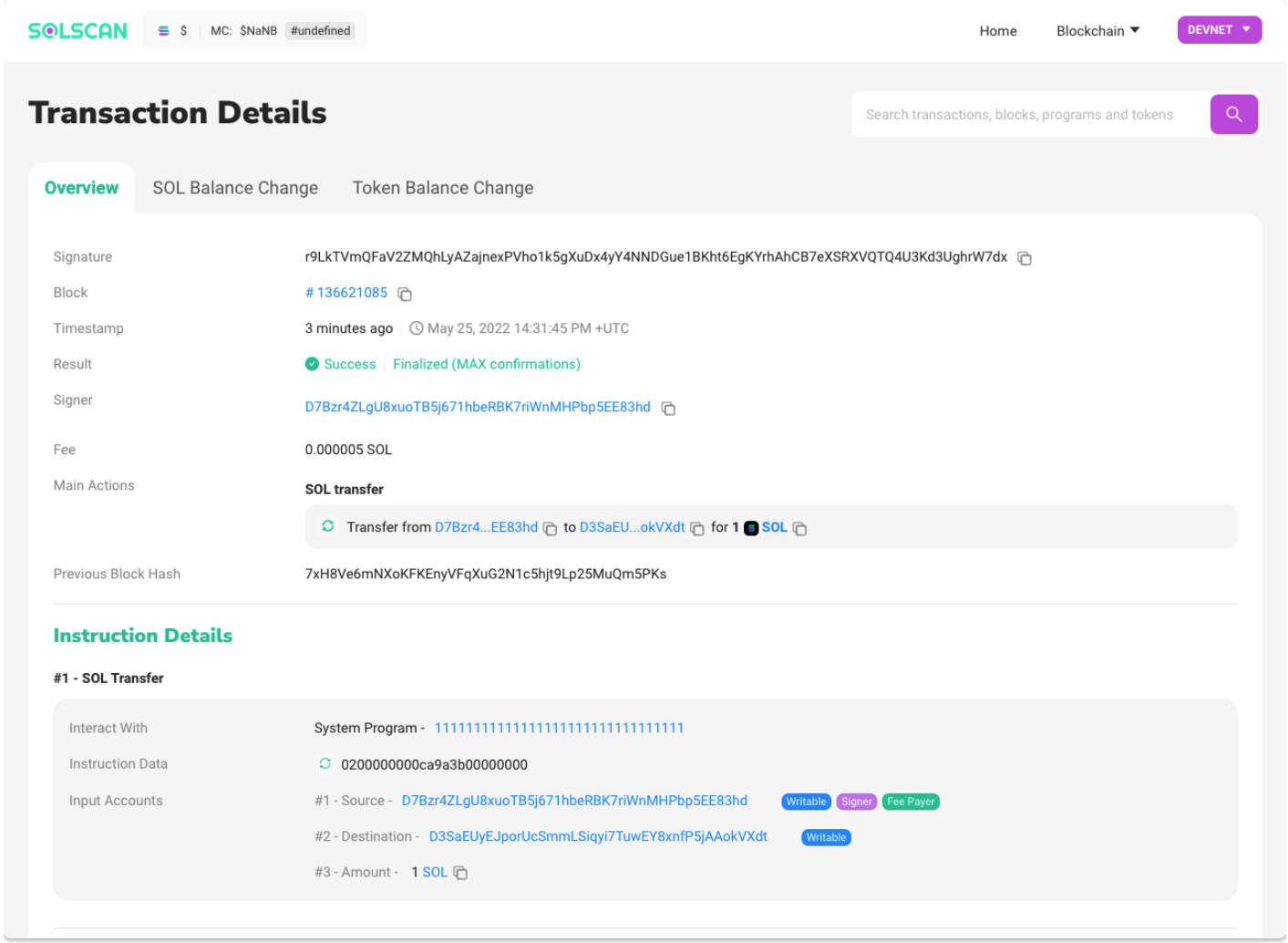

Summary
Send
1 SOL
Label
Extropy
From
Wallet 1 (D7Bz...83hd)
To
D3Sa...VXdt
Network Fee
$0.00024
Send



4. Witness confirmation on both sides



5. The transaction can be viewed using the link on the phone app


SOLSCAN
$
MC: $NaNB
#undefined
Home
Blockchain
DEVNET
Transaction Details
Search transactions, blocks, programs and tokens
Overview
SOL Balance Change
Token Balance Change
Signature
r9LkTVmQFaV2ZMQhLyAZajnexPVho1k5gXuDx4yY4NNDGue1BKht6EgKYrhAhCB7eXSRXVQTQ4U3Kd3UghrW7dx
Block
136621085
Timestamp
3 minutes ago
May 25, 2022 14:31:45 PM +UTC
Result
Success
Finalized (MAX confirmations)
Signer
D7Bzr4ZLgU8xuoTB5j671hbeRBK7riWnMHPbp5EE83hd
Fee
0.000005 SOL
Main Actions
SOL transfer
Transfer from D7Bzr4...EE83hd to D3SaEU...okVXdt for 1 SOL
Previous Block Hash
7xH8Ve6mNXoKFKEnyVFqXuG2N1c5hjt9Lp25MuQm5PKs
Instruction Details
#1 - SOL Transfer
Interact With
System Program - 11111111111111111111111111111111
Instruction Data
0200000000ca9a3b00000000
Input Accounts
#1 - Source - D7Bzr4ZLgU8xuoTB5j671hbeRBK7riWnMHPbp5EE83hd
Writable
Signer
Fee Payer
#2 - Destination - D3SaEUyEJporUcSmmLSiqyi7TuwEY8xnfP5jAAokVXdt
Writable
#3 - Amount - 1 SOL