Token Program

A Token program on the Solana blockchain.

This program defines a common implementation for Fungible and Non Fungible tokens.

Background

Solana's programming model and the definitions of the Solana terms used in this document are available at:

- https://docs.solana.com/apps
- https://docs.solana.com/terminology

Source

The Token Program's source is available onGitHub

Interface

The Token Program is written in Rust and available oncrates.io anddocs.rs .

Auto-generated C bindings are also availablehere

JavaScript bindings are available that support loading the Token Program on to a chain and issue instructions.

See theSPL Associated Token Account program for convention around wallet address to token account mapping and funding.

Status

The SPL Token program is considered complete, and there are no plans to add new functionality. There may be changes to fix important or breaking bugs.

Reference Guide

Setup

- CLI
- JS

Thespl-token command-line utility can be used to experiment with SPL tokens. Once you haveRust installed , run:

cargo install spl-token-cli Runspl-token --help for a full description of available commands.

Configuration

Thespl-token configuration is shared with thesolana command-line tool.

Current Configuration

solana config get Config File: {HOME}/.config/solana/cli/config.yml RPC URL: https://api.mainnet-beta.solana.com WebSocket URL: wss://api.mainnet-beta.solana.com/ (computed) Keypair Path: {HOME}/.config/solana/id.json

Cluster RPC URL

SeeSolana clusters for cluster-specific RPC URLs

solana config set --url https://api.devnet.solana.com

Default Keypair

SeeKeypair conventions for information on how to setup a keypair if you don't already have one.

Keypair File

solana config set --keypair {HOME}/new-keypair.json Hardware Wallet URL (SeeURL spec)

solana config set --keypair usb://ledger/

Yarn

yarn add @solana/spl-token

npm

npm install @solana/spl-token

Configuration

You can connect to different clusters usingConnection in@solana/web3.js

const web3 =

require ('@solana/web3.js') ; const connection =

new

web3 . Connection (web3 . clusterApiUrl ('devnet') ,

'confirmed') ;

Keypair

You can either get your keypair usingKeypair from@solana/web3.js , or let the user's wallet handle the keypair and usesendTransaction fromwallet-adapter

Airdrop SOL

Creating tokens and accounts requires SOL for account rent deposits and transaction fees. If the cluster you are targeting offers a faucet, you can get a little SOL for testing:

- CLI
- JS

solana airdrop 1 import

{ clusterApiUrl ,

Connection ,

Keypair ,

LAMPORTS_PER_SOL

}

from

'@solana/web3.js' ;

const payer =

Keypair . generate ( ) ;

const connection =

new

Connection (clusterApiUrl ('devnet') , 'confirmed') ;

const airdropSignature =

await connection . requestAirdrop (payer . publicKey , LAMPORTS_PER_SOL ,) ;

await connection . confirmTransaction (airdropSignature) ;

Example: Creating your own fungible token

- CLI
- JS

spl-token create-token Creating token AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM Signature: 47hsLFxWRCg8azaZZPSnQR8DNTRsGyPNfUK7jqyzgt7wf9eag3nSnewqoZrVZHKm8zt3B6gzxhr91gdQ5qYrsRG4 import

{ createMint }

from

'@solana/spl-token' ; import

{ clusterApiUrl ,

Connection ,

Keypair ,

LAMPORTS_PER_SOL

}

from

'@solana/web3.js' ;

const payer =

Keypair . generate ( ) ; const mintAuthority =

Keypair . generate ( ) ; const freezeAuthority =

Keypair . generate ( ) ;

const connection =

new

Connection (clusterApiUrl ('devnet') , 'confirmed') ;

const mint =

await

createMint (connection , payer , mintAuthority . publicKey , freezeAuthority . publicKey , 9

// We are using 9 to match the CLI decimal default exactly) ;

console . log (mint . toBase58 ( )) ; // AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM The unique identifier of the token isAQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM .

Tokens when initially created byspl-token have no supply:

- CLI
- JS

spl-token supply AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 0 const mintInfo =

await

getMint (connection , mint)

console . log (mintInfo . supply) ; // 0 Let's mint some. First create an account to hold a balance of the newAQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM token:

- CLI
- JS

spl-token create-account AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM Creating account 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi Signature: 42Sa5eK9dMEQyvD9GMHuKxXf55WLZ7tfjabUKDhNoZRAxj9MsnN7omriWMEHXLea3aYpjZ862qocRLVikvkHkyfy const tokenAccount =

await

getOrCreateAssociatedTokenAccount (connection , payer , mint , payer . publicKey)

console . log (tokenAccount . address . toBase58 ( )) ; // 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi is now an empty account:

- CLI
- JS

spl-token balance AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 0 const tokenAccountInfo =

await

getAccount (connection , tokenAccount . address)

console . log (tokenAccountInfo . amount) ; // 0 Mint 100 tokens into the account:

- CLI
- JS

spl-token mint AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100 Minting 100 tokens Token: AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM Recipient: 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi Signature: 41mARH42fPkbYn1mvQ6hYLjmJtjW98NXwd6pHqEYg9p8RnuoUsMxVd16RkStDHEzcS2sfpSEpFscrJQn3HkHzLaa await

mintTo (connection , payer , mint , tokenAccount . address , mintAuthority , 100000000000

// because decimals for the mint are set to 9) The tokensupply and accountbalance now reflect the result of minting:

- CLI
- JS

spl-token supply AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100 spl-token balance AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100 const mintInfo =

await

getMint (connection , mint)

console . log (mintInfo . supply) ; // 100

const tokenAccountInfo =

await

getAccount (connection , tokenAccount . address)

console . log (tokenAccountInfo . amount) ; // 100

Example: View all Tokens that you own

- CLI
- JS

spl-token accounts Token Balance

7e2X5oeAAJyUTi4PfSGXFLGhyPw2H8oELm1mx87ZCgwF 84 AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100 AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 0 (Aux-1*) AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 1 (Aux-2*) import

{ AccountLayout ,

TOKEN_PROGRAM_ID }

from

"@solana/spl-token" ; import

{ clusterApiUrl ,

Connection ,

PublicKey }

from

"@solana/web3.js" ;

(async

( )

=>

{

const connection =

new

Connection (clusterApiUrl ('devnet') ,

'confirmed') ;

const tokenAccounts =

await connection . getTokenAccountsByOwner (new

PublicKey ('8YLKoCu7NwqHNS8GzuvA2ibsvLrsg22YMfMDafxh1B15') , { programId :

TOKEN_PROGRAM_ID , }) ;

```
console . log ( "Token Balance" ) ; console . log ( "------------------------------------------------------------" ) ; tokenAccounts . value . forEach ( ( tokenAccount )

=>

{ const accountData =

AccountLayout . decode ( tokenAccount . account . data ) ; console . log ( ` { new

PublicKey ( accountData . mint ) }

{ accountData . amount } ` ) ; } )

} ) ( ) ;

/* Token Balance
```

```
7e2X5oeAAJyUTi4PfSGXFLGhyPw2H8oELm1mx87ZCgwF 84 AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100
AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 0 AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 1 */
```

Example: Wrapping SOL in a Token

When you want to wrap SOL, you can send SOL to an associated token account on the native mint and callsyncNative .syncNative updates theamount field on the token account to match the amount of wrapped SOL available. That SOL is only retrievable by closing the token account and choosing the desired address to send the token account's lamports.

- CLI
- JS

```
spl-token wrap 1 Wrapping 1 SOL into GJTxcnA5Sydy8YRhqvHxbQ5QNsPyRKvzguodQEaShJje Signature:
4f4s5QVMKisLS6ihZcXXPbiBAzjnvkBcp2A7KKER7k9DwJ4qjbVsQBKv2rAyBumXC1gLn8EJQhwWkybE4yJGnw2Y import

{ NATIVE_MINT , createAssociatedTokenAccountInstruction , getAssociatedTokenAddress , createSyncNativeInstruction , getAccount }

from

"@solana/spl-token" ; import

{ clusterApiUrl ,

Connection ,

Keypair ,

LAMPORTS_PER_SOL ,

SystemProgram ,

Transaction , sendAndConfirmTransaction }

from

"@solana/web3.js" ;

( async

( )

=>

{

const connection =

new

Connection ( clusterApiUrl ( 'devnet' ) ,

'confirmed' ) ;

const wallet =

Keypair . generate ( ) ;

const airdropSignature =

await connection . requestAirdrop ( wallet . publicKey , 2

*

LAMPORTS_PER_SOL , ) ;

await connection . confirmTransaction ( airdropSignature ) ;

const associatedTokenAccount =

await

getAssociatedTokenAddress ( NATIVE_MINT , wallet . publicKey )

// Create token account to hold your wrapped SOL const ataTransaction =

new

Transaction ( ) . add ( createAssociatedTokenAccountInstruction ( wallet . publicKey , associatedTokenAccount , wallet . publicKey , NATIVE_MINT ) ) ;

await

sendAndConfirmTransaction ( connection , ataTransaction ,

[ wallet ] ) ;

// Transfer SOL to associated token account and use SyncNative to update wrapped SOL balance const solTransferTransaction =

new
```

Transaction ( ) . add (SystemProgram . transfer ({ fromPubkey : wallet . publicKey , toPubkey : associatedTokenAccount , lamports :

LAMPORTS_PER_SOL }) , createSyncNativeInstruction (associatedTokenAccount))

await

sendAndConfirmTransaction (connection , solTransferTransaction ,

[wallet]) ;

const accountInfo =

await

getAccount (connection , associatedTokenAccount) ;

console . log (Native: { accountInfo . isNative } , Lamports: { accountInfo . amount }) ;

}) ( ) ; To unwrap the Token back to SOL:

- CLI
- JS

spl-token unwrap GJTxcnA5Sydy8YRhqvHxbQ5QNsPyRKvzguodQEaShJje Unwrapping GJTxcnA5Sydy8YRhqvHxbQ5QNsPyRKvzguodQEaShJje Amount: 1 SOL Recipient: vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg Signature: f7opZ86ZHKGvkJBQsJ8Pk81v8F3v1VUfyd4kFs4CABmfTnSZK5BffETznUU3tEWvzibgKJASCf7TUpDmwGi8Rmh const walletBalance =

await connection . getBalance (wallet . publicKey) ;

console . log (Balance before unwrapping 1 WSOL: { walletBalance })

await

closeAccount (connection , wallet , associatedTokenAccount , wallet . publicKey , wallet) ;

const walletBalancePostClose =

await connection . getBalance (wallet . publicKey) ;

console . log (Balance after unwrapping 1 WSOL: { walletBalancePostClose })

/ *Balance before unwrapping 1 WSOL: 997950720 Balance after unwrapping 1 WSOL: 1999985000*/ Note : Some lamports were removed for transaction fees

Example: Transferring tokens to another user

First the receiver usesspl-token create-account to create their associated token account for the Token type. Then the receiver obtains their wallet address by runningsolana address and provides it to the sender.

The sender then runs:

- CLI
- JS

spl-token transfer AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 50 vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg Transfer 50 tokens Sender: 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi Recipient: vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg Recipient associated token account: F59618aQB8r6asXeMcB9jWuY6NEx1VduT9yFo1GTi1ks

Signature: 5a3qbvoJQnTAxGPHCugibZTbSu7xuTgkxvF4EJupRjRXGgZZrnWFmKzfEzcqKF2ogCaF4QKVbAtuFx7xGwrDUcGd import

{ clusterApiUrl ,

Connection ,

Keypair ,

LAMPORTS_PER_SOL

}

from

'@solana/web3.js' ; import

{ createMint , getOrCreateAssociatedTokenAccount , mintTo , transfer }

from

'@solana/spl-token' ;

(async

( )

=>

{ // Connect to cluster const connection =

new

Connection (clusterApiUrl ('devnet') ,

'confirmed') ;

// Generate a new wallet keypair and airdrop SOL const fromWallet =

Keypair . generate ( ) ; const fromAirdropSignature =

await connection . requestAirdrop (fromWallet . publicKey ,

LAMPORTS_PER_SOL) ;

// Wait for airdrop confirmation await connection . confirmTransaction (fromAirdropSignature) ;

// Generate a new wallet to receive newly minted token const toWallet =

```
Keypair . generate ( ) ;

// Create new token mint const mint =

await

createMint ( connection , fromWallet , fromWallet . publicKey ,

null ,

9 ) ;

// Get the token account of the fromWallet address, and if it does not exist, create it const fromTokenAccount =

await

getOrCreateAssociatedTokenAccount ( connection , fromWallet , mint , fromWallet . publicKey ) ;

// Get the token account of the toWallet address, and if it does not exist, create it const toTokenAccount =

await

getOrCreateAssociatedTokenAccount ( connection , fromWallet , mint , toWallet . publicKey ) ;

// Mint 1 new token to the "fromTokenAccount" account we just created let signature =

await

mintTo ( connection , fromWallet , mint , fromTokenAccount . address , fromWallet . publicKey , 1000000000 ) ; console . log ( 'mint tx:' , signature ) ;

// Transfer the new token to the "toTokenAccount" we just created signature =

await

transfer ( connection , fromWallet , fromTokenAccount . address , toTokenAccount . address , fromWallet . publicKey , 50 ) ; } ) ( ) ;
```

Example: Transferring tokens to another user, with sender-funding

If the receiver does not yet have an associated token account, the sender may choose to fund the receiver's account.

The receiver obtains their wallet address by runningsolana address and provides it to the sender.

The sender then runs to fund the receiver's associated token account, at the sender's expense, and then transfers 50 tokens into it:

- CLI
- JS

```
spl-token transfer --fund-recipient AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 50 vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg Transfer 50 tokens Sender: 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi Recipient: vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg Recipient associated token account: F59618aQB8r6asXeMcB9jWuY6NEx1VduT9yFo1GTi1ks Funding recipient: F59618aQB8r6asXeMcB9jWuY6NEx1VduT9yFo1GTi1ks (0.00203928 SOL)

Signature: 5a3qbvoJQnTAxGPHCugibZTbSu7xuTgkxvF4EJupRjRXGgZZrnWFmKzfEzcqKF2ogCaF4QKVbAtuFx7xGwrDUcGd const signature =

await

transfer ( connection , toWallet , fromTokenAccount . address , toTokenAccount . address , fromWallet . publicKey , 50 , [ fromWallet , toWallet ] ) ;
```

Example: Transferring tokens to an explicit recipient token account

Tokens may be transferred to a specific recipient token account. The recipient token account must already exist and be of the same Token type.

- CLI
- JS

```
spl-token create-account AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM /path/to/auxiliary_keypair.json Creating account CqAxDdBRnawzx9q4PYM3wrybLHBhDZ4P6BTV13WsRJYJ Signature: 4yPWj22mbyLu5mhfZ5WATNfYzTt5EQ7LGzryxM7Ufu7QCVjTE7czZdEBqdKR7vjKsfAqsBdjU58NJvXrTqCXvfWW spl-token accounts AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM -v Account Token Balance
```

```
7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 100 CqAxDdBRnawzx9q4PYM3wrybLHBhDZ4P6BTV13WsRJYJ AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 0 (Aux-1*) spl-token transfer AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 50 CqAxDdBRnawzx9q4PYM3wrybLHBhDZ4P6BTV13WsRJYJ Transfer 50 tokens Sender: 7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi Recipient: CqAxDdBRnawzx9q4PYM3wrybLHBhDZ4P6BTV13WsRJYJ

Signature: 5a3qbvoJQnTAxGPHCugibZTbSu7xuTgkxvF4EJupRjRXGgZZrnWFmKzfEzcqKF2ogCaF4QKVbAtuFx7xGwrDUcGd spl-token accounts AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM -v Account Token Balance
```

```
7UX2i7SucgLMQcfZ75s3VXmZZY4YRUyJN9X1RgfMoDUi AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 50 CqAxDdBRnawzx9q4PYM3wrybLHBhDZ4P6BTV13WsRJYJ AQoKYV7tYpTrFZN6P5oUufbQKAUr9mNYGe1TTJC9wajM 50 (Aux-1*) import

{ getAccount , createMint , createAccount , mintTo , getOrCreateAssociatedTokenAccount , transfer }

from

"@solana/spl-token" ; import

{ clusterApiUrl ,

Connection ,

Keypair ,

LAMPORTS_PER_SOL }

from

"@solana/web3.js" ;

( async

( )
=>

{

const connection =

new

Connection ( clusterApiUrl ( 'devnet' ) ,

'confirmed' ) ;

const wallet =

Keypair . generate ( ) ; const auxiliaryKeypair =

Keypair . generate ( ) ;

const airdropSignature =

await connection . requestAirdrop ( wallet . publicKey , LAMPORTS_PER_SOL , ) ;

await connection . confirmTransaction ( airdropSignature ) ;

const mint =

await

createMint ( connection , wallet , wallet . publicKey , wallet . publicKey , 9 ) ;

// Create custom token account const auxiliaryTokenAccount =

await

createAccount ( connection , wallet , mint , wallet . publicKey , auxiliaryKeypair ) ;

const associatedTokenAccount =

await

getOrCreateAssociatedTokenAccount ( connection , wallet , mint , wallet . publicKey ) ;

await

mintTo ( connection , wallet , mint , associatedTokenAccount . address , wallet , 50 ) ;

const accountInfo =

await

getAccount ( connection , associatedTokenAccount . address ) ;

console . log ( accountInfo . amount ) ; // 50

await

transfer ( connection , wallet , associatedTokenAccount . address , auxiliaryTokenAccount , wallet , 50 ) ;

const auxAccountInfo =

await

getAccount ( connection , auxiliaryTokenAccount ) ;

console . log ( auxAccountInfo . amount ) ; // 50 } ) ( ) ;
```

Example: Create a non-fungible token

Create the token type with zero decimal place,

- CLI
- JS

spl-token create-token --decimals 0 Creating token 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z Signature: 4kz82JUey1B9ki1McPW7NYv1NqPKCod6WNptSkYqtuiEsQb9exHaktSAHJJsm4YxuGNW4NugPJMFX9ee6WA2dXts const mint =

await

createMint (connection , wallet , wallet . publicKey , wallet . publicKey , 0) ; then create an account to hold tokens of this new type:

- CLI
- JS

spl-token create-account 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z Creating account 7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM Signature: sjChze6ecaRtvuQVZuwURyg6teYeiH8ZwT6UTuFNKjrdayQQ3KNdPB7d2DtUZ6McafBfEefejHkJ6MWQEfVHLtC const associatedTokenAccount =

await

getOrCreateAssociatedTokenAccount (connection , wallet , mint , wallet . publicKey) ; Now mint only one token into the account,

- CLI
- JS

spl-token mint 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z 1 7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM Minting 1 tokens Token: 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z Recipient: 7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM Signature: 2Kzg6ZArQRCRvcoKSiievYy3sfPqGV91Whnz6SeimhJQXKBTYQf3E54tWg3zPpYLbcDexxyTxnj4QF69ucswfdY await

mintTo (connection , wallet , mint , associatedTokenAccount . address , wallet , 1) ; and disable future minting:

- CLI
- JS

spl-token authorize 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z mint --disable Updating 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z Current mint authority: vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg New mint authority: disabled Signature:

5QpykLzZsceoKcVRRFow9QCdae4Dp2zQAcjebyEWoezPFg2Np73gHKWQicHG1mqRdXu3yiZbrft3Q8JmqNRNqhwU let transaction =

new

Transaction ( ) . add (createSetAuthorityInstruction (mint , wallet . publicKey , AuthorityType . MintTokens , null)) ;

await web3 . sendAndConfirmTransaction (connection , transaction ,

[wallet]) ; Now the7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM account holds the one and only559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z token:

- CLI
- JS

spl-token account-info 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z

Address: 7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM Balance: 1 Mint: 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z Owner: vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg State: Initialized Delegation: (not set) Close authority: (not set) spl-token supply 559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z 1 const accountInfo =

await

getAccount (connection , associatedTokenAccount . address) ;

console . log (accountInfo . amount) ; // 1 const mintInfo =

await

getMint (connection , mint) ;

console . log (mintInfo) ; /*{ address: "7KqpRwzkkeweW5jQoETyLzhvs9rcCj9dVQ1MnzudirsM", mintAuthority: "559u4Tdr9umKwft3yHMsnAxohhzkFnUBPAFtibwuZD9z", supply: 1, decimals: 0, isInitialized: true, freezeAuthority: "vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg" }* /

Multisig usage

- CLI
- JS

The main difference inspl-token command line usage when referencing multisig accounts is in specifying the--owner argument. Typically the signer specified by this argument directly provides a signature granting its authority, but in the multisig case it just points to the address of the multisig account. Signatures are then provided by the multisig signer-set members specified by the--multisig-signer argument.

Multisig accounts can be used for any authority on an SPL Token mint or token account.

- Mint account mint authority:spl-token mint ...
- ,spl-token authorize ... mint ...
- Mint account freeze authority:spl-token freeze ...
- ,spl-token thaw ...
- ,spl-token authorize ... freeze ...
- Token account owner authority:spl-token transfer ...
- ,spl-token approve ...
- ,spl-token revoke ...
- ,spl-token burn ...
- ,spl-token wrap ...
- ,spl-token unwrap ...
- ,spl-token authorize ... owner ...
- Token account close authority:spl-token close ...
- ,spl-token authorize ... close ... The main difference in using multisign is specifying the owner as the multisig key, and giving the list of signers when constructing a transaction. Normally you would provide the signer that has authority to run the transaction as the owner, but in the multisig case the owner would be the multisig key.

Multisig accounts can be used for any authority on an SPL Token mint or token account.

- Mint account mint authority:createMint(/* ... */, mintAuthority: multisigKey, /* ... */)
- Mint account freeze authority:createMint(/* ... */, freezeAuthority: multisigKey, /* ... */)
- Token account owner authority:getOrCreateAssociatedTokenAccount(/* ... */, mintAuthority: multisigKey, /* ... */)
- Token account close authority:closeAccount(/* ... */, authority: multisigKey, /* ... */)

Example: Mint with multisig authority

First create keypairs to act as the multisig signer-set. In reality, these can be any supported signer, like: a Ledger hardware wallet, a keypair file, or a paper wallet. For convenience, generated keypairs will be used in this example.

- CLI
- JS

for i in (seq 3); do solana-keygen new --no-passphrase -so "signer-{i}.json"; done Wrote new keypair to signer-1.json Wrote new keypair to signer-2.json Wrote new keypair to signer-3.json const signer1 =

Keypair . generate ( ) ; const signer2 =

Keypair . generate ( ) ; const signer3 =

Keypair . generate ( ) ; In order to create the multisig account, the public keys of the signer-set must be collected.

- CLI
- JS

for i in (seq 3); do SIGNER="signer-{i}.json"; echo "SIGNER: (solana-keygen pubkey "SIGNER")"; done signer-1.json: BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ signer-2.json: DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY signer-3.json: D7ssXHrZJjfpZXsmDf8RwfPxe1BMMMmP1CtmX3WojPmG console . log (signer1 . publicKey . toBase58 ( )) ; console . log (signer2 . publicKey . toBase58 ( )) ; console . log (signer3 . publicKey . toBase58 ( )) ; /* *BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY D7ssXHrZJjfpZXsmDf8RwfPxe1BMMMmP1CtmX3WojPmG* / Now the multisig account can be created with thespl-token create-multisig subcommand. Its first positional argument is the minimum number of signers (M) that must sign a transaction affecting a token/mint account that is controlled by this multisig account. The remaining positional arguments are the public keys of all keypairs allowed (N) to sign for the multisig account. This example will use a "2 of 3" multisig account. That is, two of the three allowed keypairs must sign all transactions.

NOTE: SPL Token Multisig accounts are limited to a signer-set of eleven signers (1<=N <=11) and minimum signers must be no more thanN (1<=M <=N)

- CLI
- JS

spl-token create-multisig 2 BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ \ DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY D7ssXHrZJjfpZXsmDf8RwfPxe1BMMMmP1CtmX3WojPmG Creating 2/3 multisig 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re Signature: 2FN4KXnczAz33SAxwsuevqrD1BvikP6LUhLie5Lz4ETt594X8R7yvMZzZW2zjmFLPsLQNHsRuhQeumExHbnUGC9A const multisigKey =

await

createMultisig (connection , payer , [signer1 . publicKey , signer2 . publicKey , signer3 . publicKey] , 2) ;

console . log (Created 2/3 multisig { multisigKey . toBase58 ( ) }) ; // Created 2/3 multisig 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re Next create the token mint and receiving accountsas previously described and set the mint account's minting authority to the multisig account

- CLI
- JS

spl-token create-token Creating token 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Signature: 3n6zmw3hS5Hyo5duuhnNvwjAbjzC42uzCA3TTsrgr9htUonzDUXdK1d8b8J77XoeSherqWQM8mD8E1TMYCpksS2r

spl-token create-account 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Creating account EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC Signature: 5mVes7wjE7avuFqzrmSCWneKBQyPAjasCLYZPNSkmqmk2YFosYWAP9hYSiZ7b7NKpV866x5gwyKbbppX3d8PcE9s

spl-token authorize 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o mint 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re Updating 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Current mint authority: 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE New mint authority: 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re Signature: yy7dJiTx1t7jvLPCRX5RQWxNRNtFwvARSfbMJG94QKEiNS4uZcp3GhhjnMgZ1CaWMWe4jVEMy9zQBoUhzomMaxC const mint =

await

createMint (connection , payer , multisigKey , multisigKey , 9) ;

const associatedTokenAccount =

await

getOrCreateAssociatedTokenAccount (connection , payer , mint , signer1 . publicKey) ; To demonstrate that the mint account is now under control of the multisig account, attempting to mint with one multisig signer fails

- CLI
- JS

spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer signer-1.json Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC RPC response error -32002: Transaction simulation failed: Error processing Instruction 0: missing required signature for instruction try

{ await

mintTo (connection , payer , mint , associatedTokenAccount . address , multisigKey , 1) }

catch

(error)

{ console . log (error) ; } // Error: Signature verification failed But repeating with a second multisig signer, succeeds

- CLI
- JS

spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer signer-1.json \ --multisig-signer signer-2.json Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC Signature: 2ubqWqZb3ooDuc8FLaBkqZwzguhtMgQpgMAHhKsWcUzjy61qtJ7cZ1bfmYktKUfnbMYWTC1S8zdKgU6m4THsgspT await

mintTo (connection , payer , mint , associatedTokenAccount . address , multisigKey , 1 , [signer1 , signer2])

const mintInfo =

await

getMint (connection , mint)

console . log (Minted { mintInfo . supply } token) ; // Minted 1 token

Example: Offline signing with multisig

Sometimes online signing is not possible or desirable. Such is the case for example when signers are not in the same geographic location or when they use air-gapped devices not connected to the network. In this case, we use offline signing which combines the previous examples ofmultisig withoffline signing and anonce account .

This example will use the same mint account, token account, multisig account, and multisig signer-set keypair filenames as the online example, as well as a nonce account that we create here:

- CLI
- JS

solana-keygen new -o nonce-keypair.json ... == pubkey: Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj == solana create-nonce-account nonce-keypair.json 1 Signature: 3DALwrAAmCDxqeb4qXZ44WjpFcwVtgmJKhV4MW5qLJVtWeZ288j6Pzz1F4BmyPpnGLfx2P8MEJXmqPchX5y2Lf3r solana nonce-account Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj Balance: 0.01 SOL Minimum Balance Required: 0.00144768 SOL Nonce blockhash: 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E Fee: 5000 lamports per signature Authority: 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE const connection =

new

Connection (clusterApiUrl ('devnet') , 'confirmed' ,) ;

const onlineAccount =

Keypair . generate ( ) ; const nonceAccount =

Keypair . generate ( ) ;

const minimumAmount =

await connection . getMinimumBalanceForRentExemption (NONCE_ACCOUNT_LENGTH ,) ;

// Form CreateNonceAccount transaction const transaction =

new

Transaction ( ) . add (SystemProgram . createNonceAccount ({ fromPubkey : onlineAccount . publicKey , noncePubkey : nonceAccount . publicKey , authorizedPubkey : onlineAccount . publicKey , lamports : minimumAmount , }) ,) ;

await web3 . sendAndConfirmTransaction (connection , transaction ,

[onlineAccount , nonceAccount])

const nonceAccountData =

await connection . getNonce (nonceAccount . publicKey , 'confirmed' ,) ;

console . log (nonceAccountData) ; /*NonceAccount { authorizedPubkey: '5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE' nonce: '6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E', feeCalculator: { lamportsPerSignature: 5000 } }* / For the fee-payer and nonce-authority roles, a local hot wallet at5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE will be used.

- CLI
- JS

First a template command is built by specifying all signers by their public key. Upon running this command, all signers will be listed as "Absent Signers" in the output. This command will be run by each offline signer to generate the corresponding signature.

NOTE: The argument to the--blockhash parameter is the "Nonce blockhash:" field from the designated durable nonce account.

spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ \ --multisig-signer DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY \ --blockhash 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E \ --fee-payer 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --nonce Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj \ --nonce-authority 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --sign-only \ --mint-decimals 9 Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC

Blockhash: 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E Absent Signers (Pubkey): 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY Next each offline signer executes the template command, replacing each instance of their public key with the corresponding keypair.

spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer signer-1.json \ --multisig-signer DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY \ --blockhash 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E \ --fee-payer 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --nonce Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj \ --nonce-authority 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --sign-only \ --mint-decimals 9 Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC

Blockhash: 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E Signers (Pubkey=Signature): BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ=2QVah9XtvPAuhDB2QwE7gNaY962DhrGP6uy9zeN4sTWvY2xDUUzce6zkQeuT3xg44wsgtUw2H5Rf8pEArPSzJvHX Absent Signers (Pubkey): 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ \ --multisig-signer signer-2.json \ --blockhash 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E \ --fee-payer 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --nonce Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj \ --nonce-authority 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE \ --sign-only \ --mint-decimals 9 Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC

Blockhash: 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E Signers (Pubkey=Signature): DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY=2brZbTiCfyVYSCp6vZE3p7qCDeFf3z1JFmJHPBrz8SnWSDZPjbpjsW2kxFHkktTNkhES3y6UULqS4eaWztLW7FrU Absent Signers (Pubkey): 5hbZyJ3KRuFvdy5QBxvE9KwK17hzkAUkQHZTxPbiWffE BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ Finally, the offline signers communicate thePubkey=Signature pair from the output of their command to the party who will broadcast the transaction to the cluster. The broadcasting party then runs the template command after modifying it as follows:

1. Replaces any corresponding public keys with their keypair (--fee-payer ...
2. and--nonce-authority ...
3. in this example)
4. Removes the--sign-only
5. argument, and in the case of themint
6. subcommand,
7. the--mint-decimals ...
8. argument as it will be queried from the cluster
9. Adds the offline signatures to the template command via the--signer
10. argument

spl-token mint 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o 1 EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC \ --owner 46ed77fd4WTN144q62BwjU2B3ogX3Xmmc8PT5Z3Xc2re \ --multisig-signer BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ \ --multisig-signer DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY \ --blockhash 6DPt2TfFBG7sR4Hqu16fbMXPj8ddHKkbU4Y3EEEWrC2E \ --fee-payer hot-wallet.json \ --nonce Fjyud2VXixk2vCs4DkBpfpsq48d81rbEzh6deKt7WvPj \ --nonce-authority hot-wallet.json \ --signer BzWpkuRrwXHq4SSSFHa8FJf6DRQy4TaeoXnkA89vTgHZ=2QVah9XtvPAuhDB2QwE7gNaY962DhrGP6uy9zeN4sTWvY2xDUUzce6zkQeuT3xg44wsgtUw2H5Rf8pEArPSzJvHX \ --signer DhkUfKgfZ8CF6PAGKwdABRL1VqkeNrTSRx8LZfpPFVNY=2brZbTiCfyVYSCp6vZE3p7qCDeFf3z1JFmJHPBrz8SnWSDZPjbpjsW2kxFHkktTNkhES3y6UULqS4eaWztLW7FrU Minting 1 tokens Token: 4VNVRJetwapjwYU8jf4qPgaCeD76wyz8DuNj8yMCQ62o Recipient: EX8zyi2ZQUuoYtXd4MKmyHYLTjqFdWeuoTHcsTdJcKHC Signature: 2AhZXVPDBVBxTQLJohyH1wAhkkSuxRiYKomSSXtwhPL9AdF3wmhrrJGD7WgvZjBPLZUFqWrockzPp9S3fvzbgicy First a raw transaction is built using the nonceAccountInformation and tokenAccount key. All signers of the transaction are noted as part of the raw transaction. This transaction will be handed to the signers later for signing.

const nonceAccountInfo =

await connection . getAccountInfo (nonceAccount . publicKey , 'confirmed') ;

const nonceAccountFromInfo = web3 . NonceAccount . fromAccountData (nonceAccountInfo . data) ;

console . log (nonceAccountFromInfo) ;

const nonceInstruction = web3 . SystemProgram . nonceAdvance ({ authorizedPubkey : onlineAccount . publicKey , noncePubkey : nonceAccount . publicKey }) ;

const nonce = nonceAccountFromInfo . nonce ;

const mintToTransaction =

new

web3 . Transaction ({ feePayer : onlineAccount . publicKey , nonceInfo :

{ nonce , nonceInstruction } }) . add (createMintToInstruction (mint , associatedTokenAccount . address , multisigkey , 1 , [signer1 , onlineAccount] , TOKEN_PROGRAM_ID)

) ; Next each offline signer will take the transaction buffer and sign it with their corresponding key.

let mintToTransactionBuffer = mintToTransaction . serializeMessage ( ) ;

let onlineSIgnature = nacl . sign . detached (mintToTransactionBuffer , onlineAccount . secretKey) ; mintToTransaction . addSignature (onlineAccount . publicKey , onlineSIgnature) ;

// Handed to offline signer for signature let offlineSignature = nacl . sign . detached (mintToTransactionBuffer , signer1 . secretKey) ; mintToTransaction . addSignature (signer1 . publicKey , offlineSignature) ;

let rawMintToTransaction = mintToTransaction . serialize ( ) ; Finally, the hot wallet will take the transaction, serialize it, and broadcast it to the network.

// Send to online signer for broadcast to network await web3 . sendAndConfirmRawTransaction (connection , rawMintToTransaction) ;

JSON RPC methods

There is a rich set of JSON RPC methods available for use with SPL Token:

- getTokenAccountBalance
- getTokenAccountsByDelegate
- getTokenAccountsByOwner
- getTokenLargestAccounts
- getTokenSupply

Seehttps://docs.solana.com/apps/jsonrpc-api for more details.

Additionally the versatilegetProgramAccounts JSON RPC method can be employed in various ways to fetch SPL Token accounts of interest.

Finding all token accounts for a specific mint

To find all token accounts for theTESTpKgj42ya3st2SQTKiANjTBmncQSCqLAZGcSPLGM mint:

curl http://api.mainnet-beta.solana.com -X POST -H "Content-Type: application/json" -d ' { "jsonrpc": "2.0", "id": 1, "method": "getProgramAccounts", "params": ["TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA", { "encoding": "jsonParsed", "filters": [{ "dataSize": 165 }, { "memcmp": { "offset": 0, "bytes": "TESTpKgj42ya3st2SQTKiANjTBmncQSCqLAZGcSPLGM" } }] }] } ' The"dataSize": 165 filter selects allToken Account s, and then the"memcmp": ... filter selects based on themint address within each token account.

Finding all token accounts for a wallet

Find all token accounts owned by thevines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg user:

curl http://api.mainnet-beta.solana.com -X POST -H "Content-Type: application/json" -d ' { "jsonrpc": "2.0", "id": 1, "method": "getProgramAccounts", "params": ["TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA", { "encoding": "jsonParsed", "filters": [{ "dataSize": 165 }, { "memcmp": { "offset": 32, "bytes": "vines1vzrYbzLMRdu58ou5XTby4qAqVRLmqo36NKPTg" } }] }] } ' The"dataSize": 165 filter selects allToken Account s, and then the"memcmp": ... filter selects based on theowner address within each token account.

Operational overview

Creating a new token type

A new token type can be created by initializing a new Mint with theInitializeMint 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.

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

Once a Mint is initialized, themint_authority can create new tokens using theMintTo instruction. As long as a Mint contains a validmint_authority , the Mint is considered to have a non-fixed supply, and themint_authority can create new tokens with theMintTo instruction at any time. TheSetAuthority instruction can be used to irreversibly set the Mint's authority toNone , rendering the Mint's supply fixed. No further tokens can ever be Minted.

Token supply can be reduced at any time by issuing aBurn instruction which removes and discards tokens from an Account.

Creating accounts

Accounts hold token balances and are created using theInitializeAccount 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 theSetAuthority instruction.

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

Transferring tokens

Balances can be transferred between Accounts using theTransfer instruction. The owner of the source Account must be present as a signer in theTransfer instruction when the source and destination accounts are different.

It's important to note that when the source and destination of aTransfer are thesame , theTransfer willalways succeed. Therefore, a successfulTransfer 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. We strongly recommend that developers are careful about checking that the source and destination aredifferent before invoking aTransfer instruction from within their program.

Burning

TheBurn instruction decreases an Account's token balance without transferring to another Account, effectively removing the token from circulation permanently.

There is no other way to reduce supply on chain. This is similar to transferring to an account with unknown private key or destroying a private key. But the act of burning by usingBurn instructions is more explicit and can be confirmed on chain by any parties.

Authority delegation

Account owners may delegate authority over some or all of their token balance using theApprove instruction. Delegated authorities may transfer or burn up to the amount they've been delegated. Authority delegation may be revoked by the Account's owner via theRevoke instruction.

Multisignatures

M of N multisignatures are supported and can be used in place of Mint authorities or Account owners or delegates. Multisignature authorities must be initialized with theInitializeMultisig instruction. Initialization specifies the set of N public keys that are valid and the number M of those N that must be present as instruction signers for the authority to be legitimate.

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

Freezing accounts

The Mint may also contain afreeze_authority which can be used to issueFreezeAccount instructions that will render an Account unusable. Token instructions that include a frozen account will fail until the Account is thawed using theThawAccount instruction. TheSetAuthority instruction can be used to change a Mint'sfreeze_authority . If a Mint'sfreeze_authority is set toNone then account freezing and thawing is permanently disabled and all currently frozen accounts will also stay frozen permanently.

Wrapping SOL

The Token Program can be used to wrap native SOL. Doing so allows native SOL to be treated like any other Token program token type and can be useful when being called from other programs that interact with the Token Program's interface.

Accounts containing wrapped SOL are associated with a specific Mint called the "Native Mint" using the public keySo11111111111111111111111111111111111111112 .

These accounts have a few unique behaviors

- InitializeAccount
- sets the balance of the initialized Account to the SOL
- balance of the Solana account being initialized, resulting in a token balance
- equal to the SOL balance.
- Transfers to and from not only modify the token balance but also transfer an
- equal amount of SOL from the source account to the destination account.
- Burning is not supported
- When closing an Account the balance may be non-zero.

The Native Mint supply will always report 0, regardless of how much SOL is currently wrapped.

Rent-exemption

To ensure a reliable calculation of supply, a consistency valid Mint, and consistently valid Multisig accounts all Solana accounts holding an Account, Mint, or Multisig must contain enough SOL to be consideredrent exempt

Closing accounts

An account may be closed using theCloseAccount instruction. When closing an Account, all remaining SOL will be transferred to another Solana account (doesn't have to be associated with the Token Program). Non-native Accounts must have a balance of zero to be closed.

Non-Fungible tokens

An NFT is simply a token type where only a single token has been minted.

Wallet Integration Guide

This section describes how to integrate SPL Token support into an existing wallet supporting native SOL. It assumes a model whereby the user has a single system account as theirmain wallet address that they send and receive SOL from.

Although all SPL Token accounts do have their own address on-chain, there's no need to surface these additional addresses to the user.

There are two programs that are used by the wallet:

- SPL Token program: generic program that is used by all SPL Tokens
- SPL Associated Token Account
- program: defines
- the convention and provides the mechanism for mapping the user's wallet
- address to the associated token accounts they hold.

How to fetch and display token holdings

ThegetTokenAccountsByOwner JSON RPC method can be used to fetch all token accounts for a wallet address.

For each token mint, the wallet could have multiple token accounts: the associated token account and/or other ancillary token accounts

By convention it is suggested that wallets roll up the balances from all token accounts of the same token mint into a single balance for the user to shield the user from this complexity.

See theGarbage Collecting Ancillary Token Accounts section for suggestions on how the wallet should clean up ancillary token accounts on the user's behalf.

Associated Token Account

Before the user can receive tokens, their associated token account must be created on-chain, requiring a small amount of SOL to mark the account as rent-exempt.

There's no restriction on who can create a user's associated token account. It could either be created by the wallet on behalf of the user or funded by a 3rd party through an airdrop campaign.

The creation process is describedhere .

It's highly recommended that the wallet create the associated token account for a given SPL Token itself before indicating to the user that they are able to receive that SPL Tokens type (typically done by showing the user their receiving address). A wallet that chooses to not perform this step may limit its user's ability to receive SPL Tokens from other wallets.

Sample "Add Token" workflow

The user should first fund their associated token account when they want to receive SPL Tokens of a certain type to:

1. Maximize interoperability with other wallet implementations
2. Avoid pushing the cost of creating their associated token account on the first sender

The wallet should provide a UI that allow the users to "add a token". The user selects the kind of token, and is presented with information about how much SOL it will cost to add the token.

Upon confirmation, the wallet creates the associated token type as the describedhere .

Sample "Airdrop campaign" workflow

For each recipient wallet addresses, send a transaction containing:

1. Create the associated token account on the recipient's behalf.
2. UseTokenInstruction::Transfer
3. to complete the transfer

Associated Token Account Ownership

⚠ The wallet should never useTokenInstruction::SetAuthority to set theAccountOwner authority of the associated token account to another address.

Ancillary Token Accounts

At any time ownership of an existing SPL Token account may be assigned to the user. One way to accomplish this is with thespl-token authorize owner command. Wallets should be prepared to gracefully manage token accounts that they themselves did not create for the user.

Transferring Tokens Between Wallets

The preferred method of transferring tokens between wallets is to transfer into associated token account of the recipient.

The recipient must provide their main wallet address to the sender. The sender then:

1. Derives the associated token account for the recipient
2. Fetches the recipient's associated token account over RPC and checks that it exists
3. If the recipient's associated token account does not yet exist, the sender
4. wallet should create the recipient's associated token account as describedhere
5. .
6. The sender's wallet may choose to inform the user that as a result of account
7. creation the transfer will require more SOL than normal.
8. However a wallet that chooses to not support creating the recipient's
9. associated token account at this time should present a message to the user with enough
10. information to find a workaround to accomplish their goal
11. UseTokenInstruction::Transfer
12. to complete the transfer

The sender's wallet must not require that the recipient's main wallet address hold a balance before allowing the transfer.

Registry for token details

At the moment there exist a few solutions for Token Mint registries:

- Hard coded addresses in the wallet or dapp
- Metaplex Token Metadata. Learn more at theToken Metadata Documentation
- The deprecated token-list repo hasinstructions for creating your own metadata

A decentralized solution is in progress.

Garbage Collecting Ancillary Token Accounts

Wallets should empty ancillary token accounts as quickly as practical by transferring into the user's associated token account. This effort serves two purposes:

- If the user is the close authority for the ancillary account, the wallet can
- reclaim SOL for the user by closing the account.
- If the ancillary account was funded by a 3rd party, once the account is
- emptied that 3rd party may close the account and reclaim the SOL.

One natural time to garbage collect ancillary token accounts is when the user next sends tokens. The additional instructions to do so can be added to the existing transaction, and will not require an additional fee.

Cleanup Pseudo Steps:

1. For all non-empty ancillary token accounts, add aTokenInstruction::Transfer
2. instruction to the transfer the full token
3. amount to the user's associated token account.
4. For all empty ancillary token accounts where the user is the close authority,
5. add aTokenInstruction::CloseAccount
6. instruction

If adding one or more of clean up instructions cause the transaction to exceed the maximum allowed transaction size, remove those extra clean up instructions. They can be cleaned up during the next send operation.

Thespl-token gc command provides an example implementation of this cleanup process.

Token Vesting

There are currently two solutions available for vesting SPL tokens:

1) Bonfida token-vesting

This program allows you to lock arbitrary SPL tokens and release the locked tokens with a determined unlock schedule. Anunlock schedule is made of aunix timestamp and a tokenamount , when initializing a vesting contract, the creator can pass an array ofunlock schedule with an arbitrary size giving the creator of the contract complete control of how the tokens unlock over time.

Unlocking works by pushing a permissionless crank on the contract that moves the tokens to the pre-specified address. The recipient address of a vesting contract can be modified by the owner of the current recipient key, meaning that vesting contract locked tokens can be traded.

- Code:https://github.com/Bonfida/token-vesting
- UI:https://vesting.bonfida.com/#/
- Audit: The audit was conducted by Kudelski, the report can be foundhere

2) Streamflow Timelock

Enables creation, withdrawal, cancelation and transfer of token vesting contracts using time-based lock and escrow accounts. Contracts are by default cancelable by the creator and transferable by the recipient.

Vesting contract creator chooses various options upon creation, such as:

- SPL token and amount to be vested
- recipient
- exact start and end date
- (optional) cliff date and amount
- (optional) release frequency

Coming soon:

- whether or not a contract is transferable by creator/recipient
- whether or not a contract is cancelable by creator/recipient
- subject/memo

Resources:

- Audit: Reports can be foundhere
- andhere
- .
- Application with the UI:https://app.streamflow.finance/vesting
- JS SDK:https://npmjs.com/@streamflow/timelock
- (source
-)
- Rust SDK:https://crates.io/crates/streamflow-timelock
- (source
-)
- Program code:https://github.com/streamflow-finance/timelock Edit this page Previous Introduction Next Token-2022 Program