JavaScript Client for Solana

What is Solana-Web3.js?#

The Solana-Web3.js library aims to provide complete coverage of Solana. The library was built on top of the Solana JSON RPC API .

You can find the full documentation for the@solana/web3.js libraryhere .

Common Terminology#

Term Definition Program Stateless executable code written to interpret instructions. Programs are capable of performing actions based on the instructions provided. Instruction The smallest unit of a program that a client can include in a transaction. Within its processing code, an instruction may contain one or more cross-program invocations. Transaction One or more instructions signed by the client using one or more Keypairs and executed atomically with only two possible outcomes: success or failure. For the full list of terms, seeSolana terminology

Getting Started#

Installation#

yarn#

yarn add @solana/web3.js

npm#

npm install --save @solana/web3.js

Bundle#

Usage#

Javascript#

const solanaWeb3 = require("@solana/web3.js"); console.log(solanaWeb3);

ES6#

import * as solanaWeb3 from "@solana/web3.js"; console.log(solanaWeb3);

Browser Bundle#

// solanaWeb3 is provided in the global namespace by the bundle script console.log(solanaWeb3);

Quickstart#

Connecting to a Wallet#

To allow users to use your dApp or application on Solana, they will need to get access to their Keypair. A Keypair is a private key with a matching public key, used to sign transactions.

There are two ways to obtain a Keypair:

1. Generate a new Keypair
2. Obtain a Keypair using the secret key

You can obtain a new Keypair with the following:

const { Keypair } = require("@solana/web3.js");

let keypair = Keypair.generate(); This will generate a brand new Keypair for a user to fund and use within your application.

You can allow entry of the secretKey using a textbox, and obtain the Keypair withKeypair.fromSecretKey(secretKey) .

const { Keypair } = require("@solana/web3.js");

let secretKey = Uint8Array.from([202, 171, 192, 129, 150, 189, 204, 241, 142, 71, 205, 2, 81, 97, 2, 176, 48, 81, 45, 1, 96, 138, 220, 132, 231, 131, 120, 77, 66, 40, 97, 172, 91, 245, 84, 221, 157, 190, 9, 145, 176, 130, 25, 43, 72, 107, 190, 229, 75, 88, 191, 136, 7, 167, 109, 91, 170, 164, 186, 15, 142, 36, 12, 23,]);

let keypair = Keypair.fromSecretKey(secretKey); Many wallets today allow users to bring their Keypair using a variety of extensions or web wallets. The general recommendation is to use wallets, not Keypairs, to sign transactions. The wallet creates a layer of separation between the dApp and the Keypair, ensuring that the dApp never has access to the secret key. You can find ways to connect to external wallets with thewallet-adapter library.

Creating and Sending Transactions#

To interact with programs on Solana, you create, sign, and send transactions to the network. Transactions are collections of instructions with signatures. The order that instructions exist in a transaction determines the order they are executed.

A transaction in Solana-Web3.js is created using theTransaction object and adding desired messages, addresses, or instructions.

Take the example of a transfer transaction:

const { Keypair, Transaction, SystemProgram, LAMPORTS_PER_SOL, } = require("@solana/web3.js");

let fromKeypair = Keypair.generate(); let toKeypair = Keypair.generate(); let transaction = new Transaction();

transaction.add(SystemProgram.transfer({ fromPubkey: fromKeypair.publicKey, toPubkey: toKeypair.publicKey, lamports: LAMPORTS_PER_SOL, }),); The above code achieves creating a transaction ready to be signed and broadcasted to the network. TheSystemProgram.transfer instruction was added to the transaction, containing the amount of lamports to send, and theto andfrom public keys.

All that is left is to sign the transaction with keypair and send it over the network. You can accomplish sending a transaction by usingsendAndConfirmTransaction if you wish to alert the user or do something after a transaction is finished, or usesendTransaction if you don't need to wait for the transaction to be confirmed.

const { sendAndConfirmTransaction, clusterApiUrl, Connection, } = require("@solana/web3.js");

let keypair = Keypair.generate(); let connection = new Connection(clusterApiUrl("testnet"));

sendAndConfirmTransaction(connection, transaction, [keypair]); The above code takes in aTransactionInstruction usingSystemProgram , creates aTransaction , and sends it over the network. You useConnection in order to define which Solana network you are connecting to, namelymainnet-beta ,testnet , ordevnet .

Interacting with Custom Programs#

The previous section visits sending basic transactions. In Solana everything you do interacts with different programs, including the previous section's transfer transaction. At the time of writing programs on Solana are either written in Rust or C.

Let's look at theSystemProgram . The method signature for allocating space in your account on Solana in Rust looks like this:

pub fn allocate(pubkey: &Pubkey, space: u64) -> Instruction In Solana when you want to interact with a program you must first know all the accounts you will be interacting with.

You must always provide every account that the program will be interacting within the instruction. Not only that, but you must provide whether or not the account isisSigner orisWritable .

In theallocate method above, a single accountpubkey is required, as well as an amount ofspace for allocation. We know that theallocate method writes to the account by allocating space within it, making thepubkey required to beisWritable .isSigner is required when you are designating the account that is running the instruction. In this case, the signer is the account calling to allocate space within itself.

Let's look at how to call this instruction using solana-web3.js:

let keypair = web3.Keypair.generate(); let payer = web3.Keypair.generate(); let connection = new web3.Connection(web3.clusterApiUrl("testnet"));

let airdropSignature = await connection.requestAirdrop(payer.publicKey, web3.LAMPORTS_PER_SOL,);

await connection.confirmTransaction({ signature: airdropSignature }); First, we set up the account Keypair and connection so that we have an account to make allocate on the testnet. We also create a payer Keypair and airdrop some sol so we can

pay for the allocate transaction.

let allocateTransaction = new web3.Transaction({ feePayer: payer.publicKey, }); let keys = [{ pubkey: keypair.publicKey, isSigner: true, isWritable: true }]; let params = { space: 100 }; We create the transactionallocateTransaction , keys, and params objects.feePayer is an optional field when creating a transaction that specifies who is paying for the transaction, defaulting to the pubkey of the first signer in the transaction.keys represents all accounts that the program'sallocate function will interact with. Since theallocate function also required space, we createdparams to be used later when invoking theallocate function.

let allocateStruct = { index: 8, layout: struct([u32("instruction"), ns64("space")]), }; The above is created usingu32 andns64 from@solana/buffer-layout to facilitate the payload creation. Theallocate function takes in the parameterspace . To interact with the function we must provide the data as a Buffer format. Thebuffer-layout library helps with allocating the buffer and encoding it correctly for Rust programs on Solana to interpret.

Let's break down this struct.

{ index: 8, / <-- / layout: struct([u32('instruction'), ns64('space'),]) } index is set to 8 because the functionallocate is in the 8th position in the instruction enum forSystemProgram .

/ *https://github.com/solana-labs/solana/blob/21bc43ed58c63c827ba4db30426965ef3e807180/sdk/program/src/system_instruction.rs#L142-L305* / pub enum SystemInstruction { / **0** /CreateAccount {//**},** / 1 /**Assign {**//}, / **2** /Transfer {//**},** / 3 /**CreateAccountWithSeed {**//}, / **4** /AdvanceNonceAccount, / **5** /WithdrawNonceAccount(u64), / **6** /InitializeNonceAccount(Pubkey), / **7** /AuthorizeNonceAccount(Pubkey), / **8** /Allocate {//**},** / 9 /**AllocateWithSeed {**//}, / **10** /AssignWithSeed {//**},** / 11 /**TransferWithSeed {**//}, / **12** /UpgradeNonceAccount, } Next up isu32('instruction') .

{ index: 8, layout: struct([u32('instruction'), / <-- / ns64('space'),]) } Thelayout in the allocate struct must always haveu32('instruction') first when you are using it to call an instruction.

{ index: 8, layout: struct([u32('instruction'), ns64('space'), / <-- /]) } ns64('space') is the argument for theallocate function. You can see in the originalallocate function in Rust that space was of the typeu64 .u64 is an unsigned 64bit integer. Javascript by default only provides up to 53bit integers.ns64 comes from@solana/buffer-layout to help with type conversions between Rust and Javascript. You can find more type conversions between Rust and Javascript atsolana-labs/buffer-layout .

let data = Buffer.alloc(allocateStruct.layout.span); let layoutFields = Object.assign({ instruction: allocateStruct.index }, params); allocateStruct.layout.encode(layoutFields, data); Using the previously created bufferLayout, we can allocate a data buffer. We then assign our params{ space: 100 } so that it maps correctly to the layout, and encode it to the data buffer. Now the data is ready to be sent to the program.

allocateTransaction.add(new web3.TransactionInstruction({ keys, programId: web3.SystemProgram.programId, data, }),);

await web3.sendAndConfirmTransaction(connection, allocateTransaction, [payer, keypair,]); Finally, we add the transaction instruction with all the account keys, payer, data, and programId and broadcast the transaction to the network.

The full code can be found below.

```
const { struct, u32, ns64 } = require("@solana/buffer-layout"); const { Buffer } = require("buffer"); const web3 = require("@solana/web3.js");

let keypair = web3.Keypair.generate(); let payer = web3.Keypair.generate();

let connection = new web3.Connection(web3.clusterApiUrl("testnet"));

let airdropSignature = await connection.requestAirdrop( payer.publicKey, web3.LAMPORTS_PER_SOL, );

await connection.confirmTransaction({ signature: airdropSignature });

let allocateTransaction = new web3.Transaction({ feePayer: payer.publicKey, }); let keys = [{ pubkey: keypair.publicKey, isSigner: true, isWritable: true }]; let params = { space: 100 };

let allocateStruct = { index: 8, layout: struct([u32("instruction"), ns64("space")]), };

let data = Buffer.alloc(allocateStruct.layout.span); let layoutFields = Object.assign({ instruction: allocateStruct.index }, params); allocateStruct.layout.encode(layoutFields, data);

allocateTransaction.add( new web3.TransactionInstruction({ keys, programId: web3.SystemProgram.programId, data, }), );

await web3.sendAndConfirmTransaction(connection, allocateTransaction, [ payer, keypair, ]);
```