Web3.js API Examples

Web3 API Reference Guide#

The@solana/web3.js library is a package that has coverage over theSolana JSON RPC API .

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

General#

Connection#

Source Documentation

Connection is used to interact with theSolana JSON RPC . You can use Connection to confirm transactions, get account info, and more.

You create a connection by defining the JSON RPC cluster endpoint and the desired commitment. Once this is complete, you can use this connection object to interact with any of the Solana JSON RPC API.

Example Usage#

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

let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

let slot = await connection.getSlot(); console.log(slot); // 93186439

let blockTime = await connection.getBlockTime(slot); console.log(blockTime); // 1630747045

let block = await connection.getBlock(slot); console.log(block);

/ *{ blockHeight: null, blockTime: 1630747045, blockhash: 'AsFv1aV5DGip9YJHHqVjrGg6EKk55xuyxn2HeiN9xQyn', parentSlot: 93186438, previousBlockhash: '11111111111111111111111111111111', rewards: [], transactions: [] }* /

let slotLeader = await connection.getSlotLeader(); console.log(slotLeader); //49AqLYbpJYc2DrzGUAH1fhWJy62yxBxpLEkfJwjKy2jr The above example shows only a few of the methods on Connection. Please see thesource generated docs for the full list.

Transaction#

Source Documentation

A transaction is used to interact with programs on the Solana blockchain. These transactions are constructed with TransactionInstructions, containing all the accounts possible to interact with, as well as any needed data or program addresses. Each TransactionInstruction consists of keys, data, and a programId. You can do multiple instructions in a single transaction, interacting with multiple programs at once.

Example Usage#

const web3 = require("@solana/web3.js"); const nacl = require("tweetnacl");

// Airdrop SOL for paying transactions let payer = web3.Keypair.generate(); let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

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

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

let toAccount = web3.Keypair.generate();

// Create Simple Transaction let transaction = new web3.Transaction();

// Add an instruction to execute transaction.add(web3.SystemProgram.transfer({ fromPubkey: payer.publicKey, toPubkey: toAccount.publicKey, lamports: 1000, }),);

// Send and confirm transaction // Note: feePayer is by default the first signer, or payer, if the parameter is not set await web3.sendAndConfirmTransaction(connection, transaction, [payer]);

// Alternatively, manually construct the transaction let recentBlockhash = await connection.getRecentBlockhash(); let manualTransaction = new web3.Transaction({ recentBlockhash: recentBlockhash.blockhash, feePayer: payer.publicKey, }); manualTransaction.add(web3.SystemProgram.transfer({ fromPubkey: payer.publicKey, toPubkey: toAccount.publicKey, lamports: 1000, }),);

let transactionBuffer = manualTransaction.serializeMessage(); let signature = nacl.sign.detached(transactionBuffer, payer.secretKey);

manualTransaction.addSignature(payer.publicKey, signature);

let isVerifiedSignature = manualTransaction.verifySignatures(); console.log(The signatures were verified: {isVerifiedSignature});

// The signatures were verified: true

let rawTransaction = manualTransaction.serialize();

await web3.sendAndConfirmRawTransaction(connection, rawTransaction);

Keypair#

Source Documentation

The keypair is used to create an account with a public key and secret key within Solana. You can either generate, generate from a seed, or create from a secret key.

Example Usage#

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

let account = Keypair.generate();

console.log(account.publicKey.toBase58()); console.log(account.secretKey);

// 2DVaHtcdTf7cm18Zm9VV8rKK4oSnjmTkKE6MiXe18Qsb // Uint8Array(64) [// 152, 43, 116, 211, 207, 41, 220, 33, 193, 168, 118, // 24, 176, 83, 206, 132, 47, 194, 2, 203, 186, 131, // 197, 228, 156, 170, 154, 41, 56, 76, 159, 124, 18, // 14, 247, 32, 210, 51, 102, 41, 43, 21, 12, 170, // 166, 210, 195, 188, 60, 220, 210, 96, 136, 158, 6, // 205, 189, 165, 112, 32, 200, 116, 164, 234 //]

let seed = Uint8Array.from([70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, 100,]); let accountFromSeed = Keypair.fromSeed(seed);

console.log(accountFromSeed.publicKey.toBase58()); console.log(accountFromSeed.secretKey);

// 3LDverZtSC9Duw2wyGC1C38atMG49toPNW9jtGJiw9Ar // Uint8Array(64) [// 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, 102, // 100, 70, 60, 102, 100, 70, 60, 102, 100, 70, 60, // 102, 100, 70, 60, 102, 100, 70, 60, 102, 100, 34, // 164, 6, 12, 9, 193, 196, 30, 148, 122, 175, 11, // 28, 243, 209, 82, 240, 184, 30, 31, 56, 223, 236, // 227, 60, 72, 215, 47, 208, 209, 162, 59 //]

let accountFromSecret = Keypair.fromSecretKey(account.secretKey);

console.log(accountFromSecret.publicKey.toBase58()); console.log(accountFromSecret.secretKey);

// 2DVaHtcdTf7cm18Zm9VV8rKK4oSnjmTkKE6MiXe18Qsb // Uint8Array(64) [// 152, 43, 116, 211, 207, 41, 220, 33, 193, 168, 118, // 24, 176, 83, 206, 132, 47, 194, 2, 203, 186, 131, // 197, 228, 156, 170, 154, 41, 56, 76, 159, 124, 18, // 14, 247, 32, 210, 51, 102, 41, 43, 21, 12, 170, // 166, 210, 195, 188, 60, 220, 210, 96, 136, 158, 6, // 205, 189, 165, 112, 32, 200, 116, 164, 234 //] Usinggenerate generates a random Keypair for use as an account on Solana. UsingfromSeed , you can generate a Keypair using a deterministic constructor.fromSecret creates a Keypair from a secret Uint8array. You can see that the publicKey for thegenerate Keypair andfromSecret Keypair are the same because the secret from thegenerate Keypair is used infromSecret .

Warning : Do not usefromSeed unless you are creating a seed with high entropy. Do not share your seed. Treat the seed like you would a private key.

PublicKey#

Source Documentation

PublicKey is used throughout@solana/web3.js in transactions, keypairs, and programs. You require publickey when listing each account in a transaction and as a general identifier on Solana.

A PublicKey can be created with a base58 encoded string, buffer, Uint8Array, number, and an array of numbers.

Example Usage#

const { Buffer } = require("buffer"); const web3 = require("@solana/web3.js"); const crypto = require("crypto");

// Create a PublicKey with a base58 encoded string let base58publicKey = new web3.PublicKey("5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6Hkj",); console.log(base58publicKey.toBase58());

// 5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6Hkj

// Create a Program Address let highEntropyBuffer = crypto.randomBytes(31); let programAddressFromKey = await web3.PublicKey.createProgramAddress([highEntropyBuffer.slice(0, 31)], base58publicKey,); console.log(Generated Program Address: {programAddressFromKey.toBase58()});

// Generated Program Address: 3thxPEEz4EDWHNxo1LpEpsAxZryPAHyvNVXJEJWgBgwJ

// Find Program address given a PublicKey let validProgramAddress = await web3.PublicKey.findProgramAddress([Buffer.from("", "utf8")], programAddressFromKey,); console.log(Valid Program Address: {validProgramAddress});

// Valid Program Address: C14Gs3oyeXbASzwUpqSymCKpEyccfEuSe8VRar9vJQRE,253

SystemProgram#

Source Documentation

The SystemProgram grants the ability to create accounts, allocate account data, assign an account to programs, work with nonce accounts, and transfer lamports. You can use the SystemInstruction class to help with decoding and reading individual instructions

Example Usage#

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

// Airdrop SOL for paying transactions let payer = web3.Keypair.generate(); let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

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

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

// Allocate Account Data let allocatedAccount = web3.Keypair.generate(); let allocateInstruction = web3.SystemProgram.allocate({ accountPubkey: allocatedAccount.publicKey, space: 100, }); let transaction = new web3.Transaction().add(allocateInstruction);

await web3.sendAndConfirmTransaction(connection, transaction, [payer, allocatedAccount,]);

// Create Nonce Account let nonceAccount = web3.Keypair.generate(); let minimumAmountForNonceAccount = await connection.getMinimumBalanceForRentExemption(web3.NONCE_ACCOUNT_LENGTH); let createNonceAccountTransaction = new web3.Transaction().add(web3.SystemProgram.createNonceAccount({ fromPubkey: payer.publicKey, noncePubkey: nonceAccount.publicKey, authorizedPubkey: payer.publicKey, lamports: minimumAmountForNonceAccount, }),);

await web3.sendAndConfirmTransaction(connection, createNonceAccountTransaction, [payer, nonceAccount],);

// Advance nonce - Used to create transactions as an account custodian let advanceNonceTransaction = new web3.Transaction().add(web3.SystemProgram.nonceAdvance({ noncePubkey: nonceAccount.publicKey, authorizedPubkey: payer.publicKey, }),);

await web3.sendAndConfirmTransaction(connection, advanceNonceTransaction, [payer,]);

// Transfer lamports between accounts let toAccount = web3.Keypair.generate();

let transferTransaction = new web3.Transaction().add(web3.SystemProgram.transfer({ fromPubkey: payer.publicKey, toPubkey: toAccount.publicKey, lamports: 1000, }),); await web3.sendAndConfirmTransaction(connection, transferTransaction, [payer]);

// Assign a new account to a program let programId = web3.Keypair.generate(); let assignedAccount = web3.Keypair.generate();

let assignTransaction = new web3.Transaction().add(web3.SystemProgram.assign({ accountPubkey: assignedAccount.publicKey, programId: programId.publicKey, }),);

await web3.sendAndConfirmTransaction(connection, assignTransaction, [payer, assignedAccount,]);

Secp256k1Program#

Source Documentation

The Secp256k1Program is used to verify Secp256k1 signatures, which are used by both Bitcoin and Ethereum.

Example Usage#

const { keccak_256 } = require("js-sha3"); const web3 = require("@solana/web3.js"); const secp256k1 = require("secp256k1");

// Create a Ethereum Address from secp256k1 let secp256k1PrivateKey; do { secp256k1PrivateKey = web3.Keypair.generate().secretKey.slice(0, 32); } while (!secp256k1.privateKeyVerify(secp256k1PrivateKey));

let secp256k1PublicKey = secp256k1 .publicKeyCreate(secp256k1PrivateKey, false) .slice(1);

let ethAddress = web3.Secp256k1Program.publicKeyToEthAddress(secp256k1PublicKey); console.log(Ethereum Address: 0x{ethAddress.toString("hex")});

// Ethereum Address: 0xadbf43eec40694eacf36e34bb5337fba6a2aa8ee

// Fund a keypair to create instructions let fromPublicKey = web3.Keypair.generate(); let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

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

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

// Sign Message with Ethereum Key let plaintext = Buffer.from("string address"); let plaintextHash = Buffer.from(keccak_256.update(plaintext).digest()); let { signature, recid: recoveryId } = secp256k1.ecdsaSign(plaintextHash, secp256k1PrivateKey,);

// Create transaction to verify the signature let transaction = new Transaction().add(web3.Secp256k1Program.createInstructionWithEthAddress({ ethAddress: ethAddress.toString("hex"), plaintext, signature, recoveryId, }),);

// Transaction will succeed if the message is verified to be signed by the address await web3.sendAndConfirmTransaction(connection, transaction, [fromPublicKey]);

Message#

Source Documentation

Message is used as another way to construct transactions. You can construct a message using the accounts, header, instructions, and recentBlockhash that are a part of a transaction. ATransaction is a Message plus the list of required signatures required to execute the transaction.

Example Usage#

const { Buffer } = require("buffer"); const bs58 = require("bs58"); const web3 = require("@solana/web3.js");

let toPublicKey = web3.Keypair.generate().publicKey; let fromPublicKey = web3.Keypair.generate();

let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

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

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

let type = web3.SYSTEM_INSTRUCTION_LAYOUTS.Transfer; let data = Buffer.alloc(type.layout.span); let layoutFields = Object.assign({ instruction: type.index }); type.layout.encode(layoutFields, data);

let recentBlockhash = await connection.getRecentBlockhash();

let messageParams = { accountKeys: [fromPublicKey.publicKey.toString(), toPublicKey.toString(), web3.SystemProgram.programId.toString(),], header: { numReadonlySignedAccounts: 0, numReadonlyUnsignedAccounts: 1, numRequiredSignatures: 1, }, instructions: [{ accounts: [0, 1], data: bs58.encode(data), programIdIndex: 2, },], recentBlockhash, };

let message = new web3.Message(messageParams);

let transaction = web3.Transaction.populate(message, [fromPublicKey.publicKey.toString(),]);

await web3.sendAndConfirmTransaction(connection, transaction, [fromPublicKey]);

Struct#

Source Documentation

The struct class is used to create Rust compatible structs in javascript. This class is only compatible with Borsh encoded Rust structs.

Example Usage#

Struct in Rust:

pub struct Fee { pub denominator: u64, pub numerator: u64, } Using web3:

import BN from "bn.js"; import { Struct } from "@solana/web3.js";

export class Fee extends Struct { denominator: BN; numerator: BN; }

Enum#

Source Documentation

The Enum class is used to represent a Rust compatible Enum in javascript. The enum will just be a string representation if logged but can be properly encoded/decoded when used in conjunction withStruct . This class is only compatible with Borsh encoded Rust enumerations.

Example Usage#

Rust:

pub enum AccountType { Uninitialized, StakePool, ValidatorList, } Web3:

import { Enum } from "@solana/web3.js";

export class AccountType extends Enum {}

NonceAccount#

Source Documentation

Normally a transaction is rejected if a transaction'srecentBlockhash field is too old. To provide for certain custodial services, Nonce Accounts are used. Transactions which use arecentBlockhash captured on-chain by a Nonce Account do not expire as long at the Nonce Account is not advanced.

You can create a nonce account by first creating a normal account, then usingSystemProgram to make the account a Nonce Account.

Example Usage#

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

// Create connection let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

// Generate accounts let account = web3.Keypair.generate(); let nonceAccount = web3.Keypair.generate();

// Fund account let airdropSignature = await connection.requestAirdrop(account.publicKey, web3.LAMPORTS_PER_SOL,);

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

// Get Minimum amount for rent exemption let minimumAmount = await connection.getMinimumBalanceForRentExemption(web3.NONCE_ACCOUNT_LENGTH,);

// Form CreateNonceAccount transaction let transaction = new web3.Transaction().add(web3.SystemProgram.createNonceAccount({ fromPubkey: account.publicKey, noncePubkey: nonceAccount.publicKey, authorizedPubkey: account.publicKey, lamports: minimumAmount, }),); // Create Nonce Account await

web3.sendAndConfirmTransaction(connection, transaction, [account, nonceAccount,]);

let nonceAccountData = await connection.getNonce(nonceAccount.publicKey, "confirmed",);

console.log(nonceAccountData); // NonceAccount { // authorizedPubkey: PublicKey { // _bn: // }, // nonce: '93zGZbhMmReyz4YHXjt2gHsvu5tjARsyukxD4xnaWaBq', // feeCalculator: { lamportsPerSignature: 5000 } // }

let nonceAccountInfo = await connection.getAccountInfo(nonceAccount.publicKey, "confirmed",);

let nonceAccountFromInfo = web3.NonceAccount.fromAccountData(nonceAccountInfo.data,);

console.log(nonceAccountFromInfo); // NonceAccount { // authorizedPubkey: PublicKey { // _bn: // }, // nonce: '93zGZbhMmReyz4YHXjt2gHsvu5tjARsyukxD4xnaWaBq', // feeCalculator: { lamportsPerSignature: 5000 } // } The above example shows both how to create aNonceAccount usingSystemProgram.createNonceAccount , as well as how to retrieve theNonceAccount from accountInfo. Using the nonce, you can create transactions offline with the nonce in place of therecentBlockhash .

VoteAccount#

Source Documentation

Vote account is an object that grants the capability of decoding vote accounts from the native vote account program on the network.

Example Usage#

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

let voteAccountInfo = await connection.getProgramAccounts(web3.VOTE_PROGRAM_ID); let voteAccountFromData = web3.VoteAccount.fromAccountData(voteAccountInfo[0].account.data,); console.log(voteAccountFromData); /
VoteAccount { nodePubkey: PublicKey { _bn: }, authorizedWithdrawer: PublicKey { _bn: }, commission: 10, rootSlot: 104570885, votes: [{ slot: 104570886, confirmationCount: 31 }, { slot: 104570887, confirmationCount: 30 }, { slot: 104570888, confirmationCount: 29 }, { slot: 104570889, confirmationCount: 28 }, { slot: 104570890, confirmationCount: 27 }, { slot: 104570891, confirmationCount: 26 }, { slot: 104570892, confirmationCount: 25 }, { slot: 104570893, confirmationCount: 24 }, { slot: 104570894, confirmationCount: 23 }, ...], authorizedVoters: [{ epoch: 242, authorizedVoter: [PublicKey] }], priorVoters: [[Object], [Object]], epochCredits: [{ epoch: 179, credits: 33723163, prevCredits: 33431259 }, { epoch: 180, credits: 34022643, prevCredits: 33723163 }, { epoch: 181, credits: 34331103, prevCredits: 34022643 }, { epoch: 182, credits: 34619348, prevCredits: 34331103 }, { epoch: 183, credits: 34880375, prevCredits: 34619348 }, { epoch: 184, credits: 35074055, prevCredits: 34880375 }, { epoch: 185, credits: 35254965, prevCredits: 35074055 }, { epoch: 186, credits: 35437863, prevCredits: 35254965 }, { epoch: 187, credits: 35672671, prevCredits: 35437863 }, { epoch: 188, credits: 35950286, prevCredits: 35672671 }, { epoch: 189, credits: 36228439, prevCredits: 35950286 }, ...], lastTimestamp: { slot: 104570916, timestamp: 1635730116 } } /

Staking#

StakeProgram#

Source Documentation

The StakeProgram facilitates staking SOL and delegating them to any validators on the network. You can use StakeProgram to create a stake account, stake some SOL, authorize accounts for withdrawal of your stake, deactivate your stake, and withdraw your funds. The StakeInstruction class is used to decode and read more instructions from transactions calling the StakeProgram

Example Usage#

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

// Fund a key to create transactions let fromPublicKey = web3.Keypair.generate(); let connection = new web3.Connection(web3.clusterApiUrl("devnet"), "confirmed");

let airdropSignature = await connection.requestAirdrop(fromPublicKey.publicKey, web3.LAMPORTS_PER_SOL,); await connection.confirmTransaction({ signature: airdropSignature });

// Create Account let stakeAccount = web3.Keypair.generate(); let authorizedAccount = web3.Keypair.generate(); /*Note:*

This is the minimum amount for a stake account -- Add additional Lamports for staking For example, we add 50 lamports as part of the stake / let lamportsForStakeAccount = (await connection.getMinimumBalanceForRentExemption(web3.StakeProgram.space,)) + 50;

let createAccountTransaction = web3.StakeProgram.createAccount({ fromPubkey: fromPublicKey.publicKey, authorized: new web3.Authorized(authorizedAccount.publicKey, authorizedAccount.publicKey,), lamports: lamportsForStakeAccount, lockup: new web3.Lockup(0, 0, fromPublicKey.publicKey), stakePubkey: stakeAccount.publicKey, }); await web3.sendAndConfirmTransaction(connection, createAccountTransaction, [fromPublicKey, stakeAccount,]);

// Check that stake is available let stakeBalance = await connection.getBalance(stakeAccount.publicKey); console.log(Stake balance: {stakeBalance}); // Stake balance: 2282930

// We can verify the state of our stake. This may take some time to become active let stakeState = await connection.getStakeActivation(stakeAccount.publicKey); console.log(Stake state: {stakeState.state}); // Stake state: inactive

// To delegate our stake, we get the current vote accounts and choose the first let voteAccounts = await connection.getVoteAccounts(); let voteAccount = voteAccounts.current.concat(voteAccounts.delinquent)[0]; let votePubkey = new web3.PublicKey(voteAccount.votePubkey);

// We can then delegate our stake to the voteAccount let delegateTransaction = web3.StakeProgram.delegate({ stakePubkey: stakeAccount.publicKey, authorizedPubkey: authorizedAccount.publicKey, votePubkey: votePubkey, }); await web3.sendAndConfirmTransaction(connection, delegateTransaction, [fromPublicKey, authorizedAccount,]);

// To withdraw our funds, we first have to deactivate the stake let deactivateTransaction = web3.StakeProgram.deactivate({ stakePubkey: stakeAccount.publicKey, authorizedPubkey: authorizedAccount.publicKey, }); await web3.sendAndConfirmTransaction(connection, deactivateTransaction, [fromPublicKey, authorizedAccount,]);

// Once deactivated, we can withdraw our funds let withdrawTransaction = web3.StakeProgram.withdraw({ stakePubkey: stakeAccount.publicKey, authorizedPubkey: authorizedAccount.publicKey, toPubkey: fromPublicKey.publicKey, lamports: stakeBalance, });

await web3.sendAndConfirmTransaction(connection, withdrawTransaction, [fromPublicKey, authorizedAccount,]);

Authorized#

Source Documentation

Authorized is an object used when creating an authorized account for staking within Solana. You can designate astaker andwithdrawer separately, allowing for a different account to withdraw other than the staker.

You can find more usage of theAuthorized object underStakeProgram

Lockup#

Source Documentation

Lockup is used in conjunction with theStakeProgram to create an account. The Lockup is used to determine how long the stake will be locked, or unable to be retrieved. If the Lockup is set to 0 for both epoch and the Unix timestamp, the lockup will be disabled for the stake account.

Example Usage#

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

let account = Keypair.generate(); let stakeAccount = Keypair.generate(); let authorized = new Authorized(account.publicKey, account.publicKey); let lockup = new Lockup(0, 0, account.publicKey);

let createStakeAccountInstruction = StakeProgram.createAccount({ fromPubkey: account.publicKey, authorized: authorized, lamports: 1000, lockup: lockup, stakePubkey: stakeAccount.publicKey, }); The above code creates acreateStakeAccountInstruction to be used when creating an account with theStakeProgram . The Lockup is set to 0 for both the epoch and Unix timestamp, disabling lockup for the account.

SeeStakeProgram for more.