Lesson 10 - Web3 Introduction

Web3 Introduction

So far we have looked at writing programs and interacting with them via development tools such as the Solana playground, we will now look at how we can write client code to interact with the programs.

A starting point for this is to use a project such as Dapp Scaffold to provide us with the boilerplate code necessary

We will use the javascript API, but there is also a Rust API

Solana - Web3.js

See Docs
and package

This is built on top of the JSON-RPC API - the specification of how clients can interact with programs.

Installation

You can install the libraries via npm or yarn

`npm install --save @solana/web3.js`
or
`yarn add @solana/web3.js`

Within your javascript client code you need to require the package

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

Common functionality

Connecting to a wallet

For external wallets you can use the wallet adapter library
For file system wallets you can control the key

pair, either by generating one in the code, or allowing the user to input the secret key.

For example

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

will generate a new keypair to be used within your client

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

let secretKey = Uint8Array.from([
  30, 11, ... , 132, 53,
]);

let keypair =
Keypair.fromSecretKey(secretKey);
```

will derive the keypair from the values provided.

Sending Transactions

Once you have the wallet setup, you can create and send transactions using the `Transaction` object.

We first create the transaction object

`let transaction = new Transaction();`

and then we add the details, for example to transfer some lamports, we need to add the public keys of the sender and receiver, and the program to be invoked, in this case the SystemProgram.

```
transaction.add(
  SystemProgram.transfer({
    fromPubkey: fromKeypair.publicKey,
    toPubkey: toKeypair.publicKey,
    lamports: 8888888,
  }),
);
```

We then need to submit the transaction to the network

```
let connection = new
Connection(clusterApiUrl("testnet"));
sendAndConfirmTransaction(connection,
transaction, [keypair]);
```

Examples from the Dapp Scaffold

You need to install a wallet plugin in your browser, such as phantom

1. Clone the repo
2. In the project root directory Install the dependencies
    1. `npm install` or `yarn install`
3. Run the server
    1. `npm run dev` or `yarn dev`
    2. Open a browser to http://localhost:3000

If you prefer to use gitpod

gitpod.io/#/https://github.com/solana-labs/dapp-scaffold

It will start automatically, but if you stop it in the terminal with CTL+C , and then run

```
yarn dev
```

It will start , with instant reloads.
When prompted, open in the browser.

If you look at the code, under `src\components\` you have some typescript files for the different pieces of functionality

Have a look at the code in `RequestAirdrop` particularly the API calls to get an understanding of what is happening.

```
const onClick = useCallback(async () => {

if (!publicKey) {

console.log('error', 'Wallet not connected!');

notify({ type: 'error', message: 'error', description: 'Wallet not connected!' });

return;

}

let signature: TransactionSignature = '';

try {
```

```
signature = await
connection.requestAirdrop(publicKey,
LAMPORTS_PER_SOL);

await
connection.confirmTransaction(signature,
'confirmed');

notify({ type: 'success', message:
'Airdrop successful!', txid: signature });

getUserSOLBalance(publicKey, connection);

} catch (error: any) {

...
}
```

Public Key

See Docs
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.

```
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(

"5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6
Hkj",
);
console.log(base58publicKey.toBase58());

//
5xot9PVkphiX2adznghwrAuxGs2zeWisNSxMW6hU6H
kj

// 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:
3thxPEEz4EDWHNxo1LpEpsAxZryPAHyvNVXJEJWgBg
wJ

// 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:
C14Gs3oyeXbASzwUpqSymCKpEyccfEuSe8VRar9vJQ
RE,253
```

You can do multiple interactions within one transaction
Example from the documentation

```
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.publi
cKey, signature);

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

// The signatures were verified: true

let rawTransaction =
manualTransaction.serialize();

await
```

```
web3.sendAndConfirmRawTransaction(connecti
on, rawTransaction);
```

System Program

Interacting with the system program allows us create accounts, allocate account data, assign an account to programs, work with nonce accounts, and transfer lamports.

```
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.getMinimumBalanceForRentExempti
on(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,
]);
```

Recent Blockhash and Nonce Accounts

See Docs
A transaction includes a recent blockhash to prevent duplication and to give transactions lifetimes.
Any transaction that is completely identical to a previous one is rejected, so adding a newer blockhash allows multiple transactions to repeat the exact same action.
Transactions also have lifetimes that are defined by the blockhash, as any transaction whose blockhash is too old will be rejected.

An alternative approach is using nonce accounts.
Durable transaction nonces are a mechanism for getting around the typical short lifetime of a transaction's `recent_blockhash`. They are implemented as a Solana Program.

See Docs for more details.