EVM-Lite JS

Release [1.0.0]

Jan 21, 2020

Contents

EVM-Lite JS provides serveral modules to interact with an EVM-Lite or Monet nodes.

The current list of modules:

- `evm-lite-core` - Core module to interact with an EVM-Lite node
- `evm-lite-keystore` - Keystore management for applications
- `evm-lite-datadir` - Data directory management for applications
- `evm-lite-utils` - An aggregate of utility functions used by EVM-Lite JS modules
- `evm-lite-consensus` - Our consensus clients as well as an `IAbstractConsensus` class
- `evm-lite-client` - A simple HTTP client for EVM-Lite

The following documentation will guide you through installing and running EVM-Lite JS, as well as providing API documentation with examples.

Will also include type definitions for parameters and returns.

CHAPTER 1

Getting Started

If you need to get any EVM-Lite JS modules into your project, you can do so by running the following command depending on what package manager you are using and which module you would like to install.

For `evm-lite-core`:

NPM:

```
$ npm install evm-lite-core@1.0.0
```

Yarn:

```
$ yarn add evm-lite-core@1.0.0
```

1.1 Usage

After that, you will need to create a `Node` instance and a consensus client depending on what you are trying to connect to.

For a Monet node, we use Babble as our consensus algorithm hence we would need a Babble client.

Note: If you are using a custom node with a different consensus, you will need to write your implementation for the client and it must extend `IAbstractConsensus` from `evm-lite-consensus`

```
// The default export of `evm-lite-core` is `Node`
const { default: Node } = require('evm-lite-core');

// We also need to the Babble class
const { Babble } = require('evm-lite-consensus');

// Create a babble client
const babble = new Babble('127.0.0.1', 8080 /* default 8080 */);
// Set consensus for node in the third parameter
const node = new Node('127.0.0.1', 8080, babble);
```

CHAPTER 2

EVM-Lite Client

A simple HTTP client for EVM-Lite.

2.1 Constructor

```
constructor(host: string, port: number = 8080)
```

```
const { default: Client } = require('evm-lite-client');

const client = new Client('localhost', 8080);
```

2.2 Methods

2.2.1 `getReceipt`

Fetches a receipt for a specific transaction hash.

Definition (TS)

```
getReceipt(txHash: string): Promise<IReceipt>
```

2.2.2 `getAccount`

Fetches balance, nonce, and bytecode for a specific address.

Definition (TS)

```
getAccount(address: string): Promise<IBaseAccount>
```

2.2.3 getInfo

Fetches information about the node.

Definition (TS)

```
getInfo<T>(): Promise<T>
```

CHAPTER 3

EVM-Lite Consensus

The consensus client implementations for EVM-Lite.

3.1 Solo

An empty class is exported for convenience when using EVM-Lite in `solo` mode.

3.2 Babble

The `Babble` class exposes methods to interact with its API.

- `getBlock(index: number)` - returns a babble block by index
- `getPeers()` - returns the current list of peers
- `getGenesisPeers()` - returns the genesis peers
- `getBlocks(start: number, count?: number)` - returns a list of blocks starting at *start*. If no count is specified, will return a single block in an array.
- `getValidatorHistory()` - returns entire history of the validator set
- `getValidators(round: number)` - returns the validator set at a specific *round*

CHAPTER 4

EVM-Lite Core

This is the core module to interact with an EVM-Lite or Monet node. It exposes classes which help interact with contracts, accounts, transactions and the node itself.

There are three main objects exposed as part of this library:

- `Node`
- `Contract`
- `Account`
- `Transaction`
- `Monet` - Simple wrapper around *Node<Babble>*

Only the `Node` class has the functionality to send requests to the node.

4.1 Account

Account object allows you to sign transactions, create new keypairs and generate an `Account` object from a private key.

4.1.1 `Account.new`

```
const { Account } = require('evm-lite-core');

const account = Account.new();
```

4.1.2 `fromPrivateKey`

Generates an account object based on a private key.

Definition (TS)

```
static fromPrivateKey(privateKey: string): Account;
```

4.1.3 signTx

Signs a transaction with the respective private key.

Definition (TS)

```
signTx(tx: ITransaction): ISignedTx
```

4.2 Contract

Contract object helps abstract out the process of working with a smart contract. Using this object you can deploy and interact with functions from the contract.

It is recommended to use wrapper static functions to create and load contract objects.

4.2.1 Contract.create

```
static create<S extends IAbstractSchema>(abi: IContractABI, bytecode: string): Contract
↪<S>
```

```
const { Contract } = require('evm-lite-core');

const contract = Contract.create(ABI, BYTECODE);
```

4.2.2 Contract.load

```
static load<S extends IAbstractSchema>(abi: IContractABI, address: string): Contract
↪<S>
```

```
const { Contract } = require('evm-lite-core');

const contract = Contract.load(ABI, ADDRESS);
```

4.2.3 deployTx

Generates a transaction representing the deployment of a contract.

Definition (TS)

```
deployTx(parameters: any[], from: string, gas: number, gasPrice: number ): Transaction
```

4.2.4 `setAddressAndAddFunctions`

Will populate contract functions once an address if set.

Definition (TS)

```
setAddressAndAddFunctions(address: string): this
```

4.3 Node

A node is essentially a wrapper around a client to access EVM-Lite specific endpoints as well as the underlying consensus.

4.3.1 Constructor

The constructor for a node object takes the following values:

```
constructor(host: string, port: number = 8080, consensus?: TConsensus)
```

Where `TConsensus` is any class which extends the `IAbstractConsensus` class exposed by `evm-lite-consensus`.

More formally:

```
class Node<TConsensus extends IAbstractConsensus | undefined>
```

Example (ES5)

```
const { default: Node } = require('evm-lite-core');
const { Babble } = require('evm-lite-consensus');

const babble = new Babble('127.0.0.1', 8080);
const node = new Node('127.0.0.1', 8080, babble);
```

4.3.2 `sendTx`

Submits a transaction that mutates state to the node. Any events return by contract functions will be parsed by default for all transactions.

Definition (TS)

```
sendTx(tx: Transaction, account: Account): Promise<IReceipt>
```

4.3.3 `transfer`

Transfers the specified number of tokens to another address.

Definition (TS)

```
transfer(from: Account, to: string, value: number, gas: number, gasPrice: number):␣
↪Promise<IReceipt>
```

Example (ES5)

```
const { Account } = require('evm-lite-core');

// Create account object from private key
const account = Account.fromPrivateKey('PRIVATE_KEY');

// First parameter if of type `Account`
node.transfer(account, 'TO_ADDRESS', 1000, 100000000, 0)
    .then(receipt => console.log(receipt))
    .catch(console.log);
```

4.3.4 `callTx`

Submits a transaction that does **not** mutate state to the node.

Definition (TS)

```
callTx<R>(tx: Transaction): Promise<R>
```

4.3.5 `getAccount`

Fetches account balance, nonce, and bytecode for a specified address

Definition (TS)

```
getAccount(address: string): Promise<IBaseAccount>
```

Example (ES5)

```
node.getAccount('0x9f640e0930370ff42c9b0c7679f83d4c7f3f98cd')
    .then(account => console.log(account))
    .catch(console.log);
```

CHAPTER 5

EVM-Lite Keystore

Keystore management for any EVM-Lite applications.

5.1 Keystore

5.1.1 Constructor

```
constructor(path: string)
```

Example

```
const keystore = new Keystore('<HOME_DIR>/.evmlc/keystore');
```

5.1.2 `list`

Should list all `V3Keyfile` files in the directory specified in the constructor.

```
list(): Promise<MonikerKeystore>
```

Example

```
keystore
    .list()
    .then(console.log)
    .catch(console.log);
```

5.1.3 get

Should get a single `V3Keyfile` file.

```
get(moniker: string): Promise<V3Keyfile>
```

Example

```
keystore
    .get('moniker')
    .then(console.log)
    .catch(console.log);
```

5.1.4 create

Should create a `V3Keyfile` encrypted with the password specified and place in the directory.

```
create(moniker: string, password: string, overridePath?: string): Promise<V3Keyfile>
```

Example

```
keystore
    .create('supersecurepassword')
    .then(console.log)
    .catch(console.log);
```

5.1.5 decrypt

Should decrypt a `V3Keyfile` with the ecrypted password.

```
decrypt(keyfile: V3Keyfile, password: string): Account
```

Example

```
keystore
    .get('moniker')
    .then(keyfile => keystore.decrypt(keyfile, 'supersecurepassword'))
    .then(console.log)
    .catch(console.log);
```

CHAPTER 6

EVM-Lite Datadir

A data directory for a client-sided EVM-Lite application takes the following structure:

Example: MonetCLI (`~/Library/MONET`)

```
.
├── keystore
│   ├── node0.json
│   ├── node1.json
│
├── monetcli.toml
```

The keystore is where all keyfiles will be stored and `monetcli.toml` is the configuration file where defaults for connection details and transaction attributes are stored.

This module one default class `DataDirectory` which allows you to interact with the directory structure explained above.

6.1 DataDirectory

6.1.1 Constructor

The constructor for a `DataDirectory` object takes the following values:

```
constructor(public readonly path: string, configName: string, private readonly␣
↪keystore?: K)
```

Where `K` is any class which extends the `AbstractKeystore` class exposed by `evm-lite-keystore`.

More formally:

```
class DataDirectory<K extends AbstractKeystore>
```

Example (ES5)

```
const { default: DataDirectory } = require('evm-lite-core');
const { default: Keystore } = require('evm-lite-keystore');

const keystore = new Keystore('path/to/keystore');
const datadir = new DataDirectory('path/to/directory', 'CONFIG_NAME', keystore);
```

6.1.2 readConfig

Reads the configuration file with the name provided in the constructor for the `DataDirectory` object.

Definition (TS)

```
readConfig(): Promise<IConfiguration>
```

6.1.3 saveConfig

Saves a new configuration to the file.

Definition (TS)

```
saveConfig(schema: IConfiguration): Promise<void>
```

6.1.4 newKeyfile

Creates a new keyfile with the specified passphrase and places it in the data directories keystore folder.

Definition (TS)

```
newKeyfile(moniker: string, passphrase: string, path?: string): Promise<IV3Keyfile>
```

6.1.5 getKeyfile

Fetches a keyfile by moniker from the respective keystore directory.

Definition (TS)

```
getKeyfile(moniker: string): Promise<IV3Keyfile>
```

6.1.6 listKeyfiles

Returns an object with the key as `moniker` and the value as the JSON keyfile.

Definition (TS)

```
listKeyfiles(): Promise<IMonikerKeystore>
```

6.1.7 updateKeyfile

Updates the passphrase for a keyfile if the old passphrase is known.

Definition (TS)

```
updateKeyfile(moniker: string, oldpass: string, newpass: string): Promise<IV3Keyfile>
```

6.1.8 importKeyfile

Imports a specified keyfile to the keystore of the data directory.

Definition (TS)

```
importKeyfile(moniker: string, keyfile: IV3Keyfile): Promise<IV3Keyfile>
```

CHAPTER 7

EVM-Lite Utils

This package provides useful utility functions which help use other libraries.

7.1 Currency

One of the more important classes this package exports is the `Currency` class. This will help you convert and do any arithmetic with the underlying currency of an `evm-lite` node.

The constructor for this class either takes a `BigNumber` or a string representing the number of tokens and the denomination.

The denominations of a Token are:

```
1 / 1 000 000 000 000 000 000 000          atto        (a) 10^-18
1 / 1 000 000 000 000 000 000              femto       (f) 10^-15
1 / 1 000 000 000 000 000                  pico        (p) 10^-12
1 / 1 000 000 000 000                      nano        (n) 10^-9
1 / 1 000 000                              micro       (u) 10^-6
1 / 1 000                                  milli       (m) 10^-3
1                                          Token       (T) 1
```

7.1.1 Example (Transfer)

We want to transfer from `0xd352d81c10266ead1a0ef87db12e9ce6ba68abf5` to `0x69c68dd72d71f784682c05776b0fb6f739549395` a value of `500T` (500 Tokens).

We would first make sure we have a running node. Then send the transfer using the abstraction provided by the `Node` object.

```
// Import node object
const Node = require('evm-lite-core').default;
const { Account } = require('evm-lite-core');
```

(continues on next page)

(continued from previous page)

```
// Import currency modules
const { Currency } = require('evm-lite-utils');

// Generate account from privatekey for address␣
↪`0xd352d81c10266ead1a0ef87db12e9ce6ba68abf5`
const account = Account.fromPrivateKey('PRIVATE_KEY');

// Create node object
const node = new Node('HOST', 8080);

node.transfer(
    account,
    '0x69c68dd72d71f784682c05776b0fb6f739549395',
    // You can do this with any other denomination
    Currency.Token.times(500),
    10000000,
    0
)
    .then(console.log)
    .catch(console.log);
```

7.1.2 Example (Conversion)

We can also convert from one denomination to another very easily using the `Currency.format` method. Say we wanted to convert `300m * 200n` (300 milli * 200 nano) tokens to `T`.

```
const am1 = new Currency('300m');
const am2 = new Currency('200n');

// this can be done for any denomination
console.log(am1.times(am2).format('T'));
```

CHAPTER 8

Developers

8.1 Prerequisites

First, we will need to install `yarn` as our `npm` client.

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

8.2 Architecture

This repo is a multi-package repository with the higher level `packages/` directory containing each of the packages.

```
.
├── packages/
│   ├── core/
│   │   ├── ...
│   ├── datadir/
│   │   ├── ...
│   ├── keystore/
│   │   ├── ...
│   ├── consensus/
│   │   ├── ...
│   ├── client/
│   │   ├── ...
│   ├── utils/
│   │   ├── ...
│   ├── tsconfig.json
│   └── tsconfig.settings.json
│
├── README.md
├── lerna.json
├── package.json
```

(continues on next page)

(continued from previous page)

```
├── tslint.json
└── yarn.lock
```

The `packages/tsconfig.settings.json` is the root configuration file for typescript. Each of the packages extends this configuration file. It also contains the path mappings for the typescript compiler to locate any symlinked packages from the same repository.

8.3 Installing Dependencies

Only proceed if you have installed `yarn` and `lerna`. If you have not yet installed these, head over to the *pre-requisites* section.

In the root directory on this project run

```
yarn install
```

This will install all package dependencies including the dependencies of the each of the smaller packages.

It will also run `lerna bootstrap` automatically which will install all their dependencies and linking any cross-dependencies.

All `typescript` files will also be built for each module in their respective `dist/` directory.

8.4 Linking & Unlinking

To link all dependencies using `yarn link`, simply run `yarn linkall`. If you wish to unlink all dependencies run `yarn unlinkall`.