Lesson 10 - Starknet JS

Starknet JS

See Docs
See workshop

Starknet JS is similar to libraries such as Web3.js.
It is designed to allow interaction with contracts on Starknet, for example allowing connections to be established, transactions to be submitted and data queried.

Installation

```
$ npm install starknet
# or for pre-release features:
$ npm install starknet@next
```

API

The main areas of the API are

- Provider API - for view calls that don't need a signature
- Account
- Signer API - allows signatures, so that state can be modified
- Contract API - an object representing a contract
- Contract Factory
- Utils

Provider API

You can create a provider with

```
const provider = new starknet.Provider()
```

or if you want specify the network

```
const provider = new starknet.Provider({
sequencer: {
        network: 'mainnet-alpha' // or 'goerli-alpha'
        }
})
```

To interact with a contract we use the provider we set up

Provider methods

callContract

```
provider.callContract(call [ , blockIdentifier ]) => Promise
```

The call object has the following structure

- call.contractAddress - Address of the contract
- call.entrypoint - Entrypoint of the call (method name)
- call.calldata - Payload for the invoking method

Response

```
{
result: string[];
}
```

getTransactionReceipt

```
provider.getTransactionReceipt(txHash) => Promise
```

Response

```
{
transaction_hash: string;
status: 'NOT_RECEIVED' | 'RECEIVED' | 'PENDING' | 'ACCEPTED_ON_L2' |
'ACCEPTED_ON_L1' | 'REJECTED';
actual_fee?: string;
status_data?: string;
messages_sent?: Array<MessageToL1>;
events?: Array<Event>;
l1_origin_message?: MessageToL2;
}
```

Deploy Contract

```
provider.deployContract(payload [ , abi ]) => Promise
```

Response

```
{
transaction_hash: string;
contract_address?: string;
};
```

Wait For Transaction

`provider.waitForTransaction(txHash [ , retryInterval]) => Promise < void >`

Wait for the transaction to be accepted on L2 or L1.

Other methods

- getBlock
- getClassAt
- getStorageAt
- getTransaction
- declareContract
- waitForTransaction

A useful library is get-starknet which provides connection methods.
If you are connecting with a wallet use the connect method from the get-starknet module

```
const starknet = await connect()
// connect to the wallet
await starknet?.enable({ starknetVersion: "v4" })
const provider = starknet.account
```

Signer API

The Signer API allows you to sign transactions and messages

You can generate a key pair by using the utility functions

`ec.genKeyPair()`
or
`getKeyPair(private_key)`

The signer object is then created with

`new starknet.Signer(keyPair)`

You can then sign messages

```
signer.signMessage(data, accountAddress) => Promise
```

Code Example

```
const privateKey = stark.randomAddress();
const starkKeyPair = ec.genKeyPair(privateKey);
const starkKeyPub = ec.getStarkKey(starkKeyPair);
```

Account API

The Account object extends the Provider object
To create the account object, an account contract needs to have been deployed, see below for guide to deploy an account contract.

```
const account  = new starknet.Account(Provider, address, starkKeyPair)
```

Account Properties

```
account.address =>string
```

Account Methods

```
account.getNonce() => Promise
account.estimateFee(calls [ , options ]) => _Promise
account.execute(calls [ , abi , transactionsDetail ]) => _Promise
account.signMessage(typedData) => _Promise
account.hashMessage(typedData) => _Promise
account.verifyMessageHash(hash, signature) => _Promise
account.verifyMessage(typedData, signature) => _Promise
```

See guide to creating and deploying an account

Contract Factory

Contract Factory allow you to deploy contracts to StarkNet.

Create the contract factory with

`new starknet.ContractFactory( compiledContract , providerOrAccount, [ , abi ] )`

Methods

contractFactory.attach(address) ⇒ *Contract*

contractFactory.deploy(constructorCalldata, addressSalt) ⇒ _Promise < Contract >

Contract

Creating the contract object

```
new starknet.Contract(abi, address, providerOrAccount)

contract.attach(address)` for changing the address of the connected
contract

contract.connect(providerOrAccount)` for changing the provider or account
```

Contract Properties

```
contract.address => string
contract.providerOrAccount => ProviderInterface | AccountInterface
contract.deployTransactionHash => string | null
contract.abi => Abi
```

Contract Interaction

1. View Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < Result >

The type of the result depends on the ABI.
The result object will be returned with each parameter available positionally and if the parameter is named, it will also be available by its name.

The override can identify the block : `overrides.blockIdentifier`

Code Example

`const bal = await contract.get_balance()`

2. Write Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < AddTransactionResponse >

Overrides can be

- overrides.signature - Signature that will be used for the transaction
- overrides.maxFee - Max Fee for the transaction
- overrides.nonce - Nonce for the transaction

Code Example

`await contract.increase_balance(13)`

Utils

Useful Methods

- toBN

  `toBN(number: BigNumberish, base?: number | 'hex'): BN`

  Converts BigNumberish to BN.
  Returns a BN.

- uint256ToBN

  `uint256ToBN(uint256: Uint256): BN`

  Function to convert `Uint256` to `BN` (big number), which uses the `bn.js` library.

- getStarkKey

  `getStarkKey(keyPair: KeyPair): string`

  Public key defined over a Stark-friendly elliptic curve that is different from the standard Ethereum elliptic curve

- getKeyPairFromPublicKey

  `getKeyPairFromPublicKey(publicKey: BigNumberish): KeyPair`

  Takes a public key and casts it into `elliptic` KeyPair format.
  Returns keyPair with public key only, which can be used to verify signatures, but can't sign anything.

- sign

  `sign(keyPair: KeyPair, msgHash: string): Signature`

  Signs a message using the provided key.
  keyPair should be an KeyPair with a valid private key.
  Returns an Signature.

- verify

  `verify(keyPair: KeyPair | KeyPair[], msgHash: string, sig: Signature): boolean`

Verifies a message using the provided key.
keyPair should be an KeyPair with a valid public key.
sig should be an Signature.
Returns true if the verification succeeds.

Example Usages

Deploy an ERC20 contract

```
const compiledErc20 = json.parse(
  fs.readFileSync("./ERC20.json").toString("ascii")
);
const erc20Response = await defaultProvider.deployContract({
  contract: compiledErc20,
  constructorCalldata: [encodeShortString('TokenName'),
encodeShortString('TokenSymbol'), recipient], // Here the `recipient`
receives the initial 1000 tokens
});

console.log("Waiting for Tx to be Accepted on Starknet - ERC20
Deployment...");
await defaultProvider.waitForTransaction(erc20Response.transaction_hash);
```

Mint Tokens

```
erc20.connect(account);

const { transaction_hash: mintTxHash } = await erc20.mint(
  account.address,
 [ "1000", "0"]
  {
    maxFee: "1"
  }
);

console.log(`Waiting for Tx to be Accepted on Starknet - Minting...`);
await defaultProvider.waitForTransaction(mintTxHash);
```

Further ERC20 interaction

```
import fs from "fs";

// Install the latest version of starknet with npm install starknet@next
and import starknet
import {
  Contract,
  Account,
  defaultProvider,
  ec,
  encode,
  hash,
  json,
  number,
  stark,
} from "starknet";
import { transformCallsToMulticallArrays } from
"./node_modules/starknet/utils/transaction.js";

// TODO: Change to OZ account contract
console.log("Reading Argent Account Contract...");
const compiledArgentAccount = json.parse(
  fs.readFileSync("./ArgentAccount.json").toString("ascii")
);
console.log("Reading ERC20 Contract...");
const compiledErc20 = json.parse(
  fs.readFileSync("./ERC20.json").toString("ascii")
);

// Since there are no Externally Owned Accounts (EOA) in StarkNet,
// all Accounts in StarkNet are contracts.

// Unlike in Ethereum where a account is created with a public and private
key pair,
// StarkNet Accounts are the only way to sign transactions and messages,
and verify signatures.
// Therefore a Account - Contract interface is needed.

// Generate public and private key pair.
const privateKey = stark.randomAddress();

const starkKeyPair = ec.genKeyPair(privateKey);
const starkKeyPub = ec.getStarkKey(starkKeyPair);

// Deploy the Account contract and wait for it to be verified on StarkNet.
console.log("Deployment Tx - Account Contract to StarkNet...");
const accountResponse = await defaultProvider.deployContract({
  contract: compiledArgentAccount,
  addressSalt: starkKeyPub,
});

// Wait for the deployment transaction to be accepted on StarkNet
console.log(
  "Waiting for Tx to be Accepted on Starknet - Argent Account
Deployment..."
);
await
defaultProvider.waitForTransaction(accountResponse.transaction_hash);

const accountContract = new Contract(
  compiledArgentAccount.abi,
  accountResponse.address
);

// Initialize argent account
console.log("Invoke Tx - Initialize Argnet Account...");
const initializeResponse = await accountContract.initialize(starkKeyPub,
"0");
console.log(
  "Waiting for Tx to be Accepted on Starknet - Initialize Account..."
);
await
defaultProvider.waitForTransaction(initializeResponse.transaction_hash);

// Use your new account address
const account = new Account(
  defaultProvider,
  accountResponse.address,
  starkKeyPair
);

// Deploy an ERC20 contract and wait for it to be verified on StarkNet.
console.log("Deployment Tx - ERC20 Contract to StarkNet...");
const erc20Response = await defaultProvider.deployContract({
  contract: compiledErc20,
});

// Wait for the deployment transaction to be accepted on StarkNet
console.log("Waiting for Tx to be Accepted on Starknet - ERC20
Deployment...");
await defaultProvider.waitForTransaction(erc20Response.transaction_hash);

// Get the erc20 contract address
const erc20Address = erc20Response.address;

// Create a new erc20 contract object
```

```
const erc20 = new Contract(compiledErc20.abi, erc20Address);

// Mint 1000 tokens to account address
console.log(`Invoke Tx - Minting 1000 tokens to ${account.address}...`);
const { transaction_hash: mintTxHash } = await erc20.mint(
  account.address,
  "1000"
);

// Wait for the invoke transaction to be accepted on StarkNet
console.log(`Waiting for Tx to be Accepted on Starknet - Minting...`);
await defaultProvider.waitForTransaction(mintTxHash);

// Check balance - should be 1000
console.log(`Calling StarkNet for account balance...`);
const balanceBeforeTransfer = await erc20.balance_of(account.address);

console.log(
  `account Address ${account.address} has a balance of:`,
  number.toBN(balanceBeforeTransfer.res, 16).toString()
);

// Execute tx transfer of 10 tokens
console.log(`Invoke Tx - Transfer 10 tokens back to erc20 contract...`);
const { code, transaction_hash: transferTxHash } = await account.execute(
  {
    contractAddress: erc20Address,
    entrypoint: "transfer",
    calldata: [erc20Address, "10"],
  },
  undefined,
  { maxFee: "0" }
);

// Wait for the invoke transaction to be accepted on StarkNet
console.log(`Waiting for Tx to be Accepted on Starknet - Transfer...`);
await defaultProvider.waitForTransaction(transferTxHash);

// Check balance after transfer - should be 990
console.log(`Calling StarkNet for account balance...`);
const balanceAfterTransfer = await erc20.balance_of(account.address);

console.log(
  `account Address ${account.address} has a balance of:`,
  number.toBN(balanceAfterTransfer.res, 16).toString()
);
```

Other Frameworks

Cairopal

repo

Starknet React

This is a collection of React hooks for StarkNet.
It is inspired by wagmi, powered by starknet.js.
repo

Vue and starknet boilerplate

https://github.com/dontpanicdao/vue-stark-boil

Starknet rs

Starknet Rust library

Starknet py

Starknet python library

Useful Tutorial from Darlington