Web3js Part 1

Introduction

> *“web3.js is a collection of libraries which allow you to interact with a local or remote ethereum node, using a HTTP/ WS or IPC connection.”*

Web3js allows developers to interact with the Ethereum blockchain and make clients to send transactions, read contract storage data, get user accounts etc.

It facilitates a connection to Ethereum nodes in order to interact with the blockchain.

Communication is via JSON RPC (Remote Procedure Call), which allows developers to view and write data on the Ethereum blockchain. JSON RPC is a textual encoding format allowing running processes to receive data.

Webjs documentation

https://web3js.readthedocs.io/en/v1.3.4/index.html (https://web3js.readthedocs.io/en/v1.3.4/index.html)

Web3js has several modules for the ethereum ecosystem.

Most common modules:

`web3.eth` - the main module that developers interact with.

`web3.utils` - utilies ie converting Wei to Ether.

Other modules:

`Net` - for interacting with network properties.

`ssh` - for interacting with the whisper protocol.

`bzz` - swarm network.

`personal` - Ethereum accounts.

Connecting to the network : Create a Web3 provider

```
import Web3 from 'web3';
const web3 = new Web3('http://127.0.0.1:8545');
// connecting to local host

const ABI = require('FILE PATH TO ABI');
const CONTRACT_ADDRESS = 'CONTRACT ADDRESS';

const myContract = new Web3.Contract(ABI, CONTRACT_ADDRESS);
```

View and write calls

All calls are made in view or write calls.

Write calls are transaction calls, which aim to update the state of the blockchain and will result in a transaction fee.

View calls are simply to read contract data on the blockchain. They don’t cost gas.

View call example

View calls are made through the `call` method in web3js.

If we want to call the ERC20 standard ***balanceOf*** function in our contract

```
function getBalance() async {

const myBalance = await myContract.methods.balanceOf(myAddress).call();

return myBalance;

};
```

The client sends a view call via `call` which is received by the RPC node. The RPC node obtains the information i.e. the required balance from the blockchain and returns the balance amount back to the client.

The syntax for a view call is

```
myContract.methods.myMethod([param1[, param2[, ...]]]).call
(options [, defaultBlock] [, callback])
```

View Call Parameters

- options - Object (optional): The options used for calling.

    - from - String (optional): The address the call “transaction” should be made from. For calls the from property is optional however it is highly recommended to explicitly set it or it may default to address(0) depending on your node or provider.
    - gasPrice - String (optional): The gas price in wei to use for this call “transaction”.
    - gas - Number (optional): The maximum gas provided for this call “transaction” (gas limit).
- defaultBlock - Number|String (optional): If you pass this parameter it will not use the default block set with contract.defaultBlock. Pre-defined block numbers as “earliest”, “latest”, and “pending” can also be used. Useful for requesting data from or replaying transactions in past blocks.
- callback - Function (optional): This callback will be fired with the result of the smart contract method execution as the second argument, or with an error object as the first argument.

Write call example

Write calls are made through the `send` method.

```
function makeTransfer() async {

const transactionHash = await myContract.methods.transfer(receiverAddress,
.send({from: myAddress});

return transactionHash;

};
```

The send method requires the sender’s address (additional options are explained below). The client sends a write call via `send` which is received by the RPC node.

There are additional options in the `send` method: `gas` and `nonce`

- gas provides a maximum gas limit for the transaction.
- nonce provides the transaction with the latest nonce.

Unlike view calls, the RPC does not return human-readable information regarding the successful transaction. It instead returns a **transaction hash**. The reason for this is because it cannot send an immediate response back to the client regarding the status of the transaction due to the average block time (see below).

All calls to the network are asynchronous

When a transaction is sent to the blockchain, it is only logged / an event is only emitted by the smart contract once the transaction has been mined to a block. Currently, the average block time on the Ethereum mainnet is around 13 seconds
https://etherscan.io/chart/blocktime (https://etherscan.io/chart/blocktime)
The usual method for getting the result of our transaction is via events.

Events

We need to subscribe to events. Why? The `send` method returns the transaction hash, but this isn't useful information for a user interface. Events provide human-readable and searchable information about a successful event.

When a transaction is submitted to the blockchain, an event is submitted to confirm that it was successful. When a user subscribes to these events, they receive information about the transaction after it is recorded on the blockchain.

There are three ways to subscribe to events:

1. Get past events

```
function getPastEvents() async {

const allPastEvents = await myContract.getPastEvents('Transfer');

return allPastEvents;

};
```

Let's say `myContract` is an ERC20 contract. When a ERC20 transaction is submitted, a Transfer event is emitted by the contract.

Web3js includes a `getPastEvents` method in the contract's methods. When called, the RPC node searches the blockchain logs to obtain the specified past events (in this case, Transfer events).

The method has an options input. The size of the blockchain, particularly the Ethereum blockchain, means that it can be quite slow to return data. Therefore options can be used to narrow the search. Here's an example:

```
const options = {

    filter: {
        value: ['1000', '1200']
        // Only get events where transfer value was 1000 or 1200
    },
    fromBlock: 0,
    toBlock: 'latest'
    // This could be a block number, or "earliest", or "pending"
};

```

2. Contract events

Another way to find events is from the `events` property in `myContract`

```
myContract.events.Transfer(options)
    .on('data', event => console.log(event))
    .on('changed', changed => console.log(changed))
    .on('error', err => throw err)
    .on('connected', str => console.log(str))
```

This method returns an event emitter:

`data` – Will fire each time an event of the type you are listening for has been emitted.

`changed` – Will fire for each event of the type you are listening for that has been removed from the blockchain.

`error` – Will fire if an error in the event subscription occurs.

`connected` – Will fire when the subscription has successfully established a connection. It will return a subscription id. This event only fires once.

Example of an event log:

```
[{
  "address": "0xe381C25de995d62b453aF8B931aAc84fcCaa7A62",
  "blockNumber": 15048427,
  "transactionHash": "0xf81d8bd942ba647cc44c398befe5a15537e398gr76hta78b567dd94
  "transactionIndex": 129,
  "blockHash": "0xe314c8962e9c80b13081ef4cdc5f320oot7684cb2df7532df603713c71f28
  "logIndex": 349,
  "removed": false,
  "id": "log_c987edef",
  "returnValues": {
    "0": "0xF9F6Rbf7ec2703ede176cC98A2276fA1F618A1b1",
    "1": "0x0000000000000000000000000000000000000000",
    "2": "20000000000000000000000",
    "from": "0xF9F646Eb5c2703ede176cC98A2276fA1F618A1b1",
    "to": "0x0000000000000000000000000000000000000000",
    "value": "20000000000000000000000"
  },
  "event": "Transfer",
  "signature": "0xddf252ad1be2c89b69c2b06rthf76haa952ba7f163c4a11628f55a4df523b
  "raw": {
    "data": "0x00000000000000000000000000000000000000000000152d02c7e14af6888880
    "topics": [
      "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
      "0x000000000000000000000000f9f613bdec2703ede176cc98a2276fa1f618a1b1",
      "0x0000000000000000000000000000000000000000000000000000000000000000"
    ]
  }
}]
```


3. The eth subscribe method

This method can be used to listen to all event logs.

```
const options = {
    fromBlock: 0,
    address: ['myAddress', 'address2'],
    // Get events from specific addresses
    topics: []
    // What topics to subscribe to
};

const subscription = web3.eth.subscribe('logs', options, (err,event) => {
    if (!err)
    console.log(event)
});

subscription.on('data', event => console.log(event))
subscription.on('changed', changed => console.log(changed))
subscription.on('error', err => { throw err })
subscription.on('connected', nr => console.log(nr))
```

Note: topics are 32-byte (256 bit) “words” that are used to describe what’s going on in an event. Topics are an advanced topic, we will cover those in the advanced EVM lessons.

Example of an event log:

```
{
  address: '0x0000000000000000000000000000000020201010',

  topics: [
    '0x4dfe1bbbcf077ddc3e01291eea2d5c70jnrg422b415d95645b9adcfd678cb1d63',
    '0x0000000000000000000000000000000000000000000000000000000000001010',
    '0x000000000000000000000032708b588tbb4656aa4ec83e4af16fc203adcb2c',
    '0x000000000000000000000001ca971963bdb4ba2bf337c90660674acff5beb3f'
  ],
  data: '0x0000000000000000000000000000000000000000000001088ecbd7b3c000000
  blockNumber: 0,
  transactionHash: '0xae884ba7e7f64bdurhte45d6eeae00bf688de7ca3d672d2a7f5b06a27
  transactionIndex: 6,
  blockHash: '0x279ccf819050d18eb1938774faa6d1734eeef8ty4ee6d0489d91bd805c7a0d8
  logIndex: 6,
  removed: false,
  id: 'log_7b5g1a97'
}
```

Other useful methods

`web3.eth.accounts` - for interacting with accounts.

`web3.eth.getAccounts` - a useful method to get a list of accounts.

`web3.eth.estimateGas` - returns the estimate gas amount for the cost of a transaction.

`web3.utils.sha3` and `web3.utils.keccack256` - functions which calculate the hash of an input.

`web3.eth.getBlockNumber` - returns the current block number