Web3.js Quick Guide

Contents

Using Web3 on Hardhat 2
Web3: 2
Provider: 2
web3.eth: 2
web3.eth.subscribe 3
web3.eth.Contract 4
Events: 5
web3.eth.accounts 6
web3.eth.abi 7
web3.utils 7

Web3.js docs: https://web3js.readthedocs.io/en/v1.7.4/#

Using Web3 on Hardhat

Installation and usage of hardhat-web3:

npm install --save-dev @nomiclabs/hardhat-web3 web3

And add the following statement to your hardhat.config.js:

require("@nomiclabs/hardhat-web3");

This plugin adds the following elements to the HardhatRuntimeEnvironment:

Web3: The Web3.js module.
web3: An instantiated Web3.js object connected to the selected network.

Web3:

Most Ethereum-supported browsers like MetaMask have an EIP-1193 compliant provider available at window.ethereum. "Web3.givenProvider" will be set if in an Ethereum supported browser.

Provider:

```
var Web3 = require('web3');

//use a web3 instance configured with the Metamask provider or a local node - eg: Ganache
var web3 = new Web3(Web3.givenProvider || "http://localhost:8545");

//use a web3 instance configured with a third party provider - eg: Alchemy
var web3 = new Web3("https://eth-mainnet.alchemyapi.io/v2/your-api-key");

//use a web3 instance configured with a websocket provider
var web3 = new Web3.providers.WebsocketProvider('ws://remotenode.com:8546'));

//create a standalone provider
let myProvider = new Web3.providers.HttpProvider("http://localhost:8545")

//set a different provider - globally or specifically on web3.eth
web3.setProvider(myProvider)
web3.eth.setProvider(myProvider)
```

web3.eth:

```
//get all accounts provided by the node
```

web3.eth.getAccounts(console.log);

//default address is used as the default "from" property, if no "from" property is specified
web3.eth.defaultAccount = '0x11f4...'

web3.eth.defaultChain = 'goerli' //mainnet, rinkeby...

// Returns the current gas price
web3.eth.getGasPrice([callback])

// Returns the current block number.
web3.eth.getBlockNumber()

// Get the balance of an address at a given block.
web3.eth.getBalance(address)

// Returns the receipt of a transaction by transaction hash.
// Note: The receipt is not available for pending transactions and returns null
web3.eth.getTransactionReceipt(hash [, callback])

web3.eth.sendTransaction(transactionObject)

transactionObject: from, to, value, gas, gasPrice, maxFeePerGas, maxPriorityFeePerGas...
data: Either a ABI byte string containing the data of the function call on a
contract, or in the case of a contract-creation transaction the initialisation code.

web3.eth.sendTransaction({
from: '0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe',
to: '0x11f4d0A3c12e86B4b5F39B213F7E19D048276DAe',
value: '1000000000000000'
})

// Executes a message call transaction, which is directly executed in the VM
web3.eth.call(transactionObject)

// This method will request/enable the accounts from the current environment. This method
// will only work if you’re using the injected provider from a application like Metamask
web3.eth.requestAccounts()

web3.eth.subscribe

The web3.eth.subscribe function lets you subscribe to events on the blockchain. It returns a subscription instance => subscription.id

on("data"): Fires on each incoming log with the log object as argument.

web3.eth.subscribe('logs', options):

Subscribes to incoming logs, filtered by the given options.

Options:

- fromBlock - Number: The fromBlock dictates at which block the subscription will start from
- address - String|Array: An address or a list of addresses to only get logs from particular account(s).
- topics - Array: An array of values which must each appear in the log entries. The order is important, if you want to leave topics out use null, e.g. [null, '0x00...'].

```
var subscription = web3.eth.subscribe('logs', {
address: '0x123456..',
topics: ['0x12345...']  }
.on("data", function(log){console.log(log);})
```

web3.eth.Contract

When you create a new contract object you give it the json interface of the respective smart contract and web3 will auto convert all calls into low level ABI calls over RPC for you.

```
const contract = new web3.eth.Contract(abi [, address][, options])
```

Options:
• address - String: The address where the contract is deployed. See options.address.
• jsonInterface - Array: The json interface of the contract. See options.jsonInterface.
• data - String: The byte code of the contract. Used when the contract gets deployed.
• from - String: The address transactions

```
var myContract = new web3.eth.Contract([...],
'0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe', {
from: '0x1234567890123456789012345678901234567891',
gasPrice: '20000000000'
});


// This default address is used as the default "from" property
contract.defaultAccount


myContract.deploy(options)
```

options - Object: The options used for deployment.
• data - String: The byte code of the contract.
• arguments - Array

```
//Will call a “constant” method and execute it in the EVM without sending any transaction
myContract.methods.myMethod(123).call()
```

// Will send a transaction to the smart contract and execute its method.
myContract.methods.myMethod([param1[, param2[, ...]]]).send(options[, callback])

myContract.methods.myMethod(123).send({from:'0xde0B295669a9FD...'})
.on('transactionHash', function(hash){...})
.on('receipt', function(confirmationNumber, receipt){...})

options:
• from - String: The address the transaction should be sent from.
• gasPrice - String (optional): The gas price in wei to use for this transaction.
• gas - Number (optional): The maximum gas provided for this transaction (gas limit).
• value - (optional): The value transferred for the transaction in wei.
• nonce - Number (optional): the nonce number of transaction

Returns:
• "transactionHash": Fired when the txn hash is available.
• "receipt": Fired when the transaction receipt is available. Receipts from contracts will have no logs property, but instead an events property with event names as keys

// Encodes the ABI for this method - 32-bit function signature hash
// plus the passed parameters in tightly packed format
myContract.methods.myMethod([param1[, param2[, ...]]]).encodeABI()

Events:

// Subscribe to an event.
myContract.events.MyEvent([options][, callback])

options:
• filter - Object (optional): Let you filter events by indexed parameters, e.g. {filter: {myNumber: [12,13]}} => get all events where “myNumber” is 12 or 13.
• fromBlock - (optional): The block number. Pre-defined block numbers as "earliest", "latest" and "pending" can also be used.
• topics - Array (optional): This allows to manually set the topics for the event filter.

myContract.events.MyEvent({
filter: {myIndexedParam: [20,23], myOtherIndexedParam: '0x123456789...'},
fromBlock: 0
}, function(error, event){ console.log(event); })
.on('data', function(event){ console.log(event); })

web3.eth.accounts

The web3.eth.accounts contains functions to generate Ethereum accounts and sign transactions.

// Signs an Ethereum transaction with a given private key.
web3.eth.accounts.signTransaction(tx, privateKey [, callback]);

tx - Object:
• nonce - String: (optional). Default will use web3.eth.getTransactionCount().
• chainId - String: (optional). Default will use web3.eth.getChainId().
• to - String: (optional) The recevier of the transaction, can be empty when deploying a contract.
• data - String: (optional) The call data of the transaction, can be empty for simple value transfers.
• value - String: (optional) The value of the transaction in wei.
• maxFeePerGas
• maxPriorityFeePerGas

web3.eth.accounts.signTransaction({
to: '0xF0109fC8DF283027b6285cc889F5aA624EaC1F55',
value: '1000000000', ...
}, '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318')
.then(console.log);

web3.eth.accounts.hashMessage("Hello World")

// Signs arbitrary data.
web3.eth.accounts.sign(data, privateKey);

Returns the signature object:
• message - String: The given message.
• messageHash - String: The hash of the given message.
• r - String: First 32 bytes of the signature
• s - String: Next 32 bytes of the signature
• v - String:

// Recovers the Ethereum address which was used to sign the given data
web3.eth.accounts.recover({messageHash: '0x1da44b586...',
v: '0x1c',
r: '0xb91467e570a6466aa9e9876cbcd013baba02900b8979d43fe208a4a4f339f5fd',
s: '0x6007e74cd82e037b800186422fc2da167c747ef045e5d18a5f5d4300f8e1a029' })

web3.eth.abi

The web3.eth.abi functions let you encode and decode parameters to ABI for function calls to the EVM

// Encodes the function name to its ABI signature, which are the first 4 bytes of the sha3 hash
web3.eth.abi.encodeFunctionSignature('myMethod(uint256,string)')

// Encodes the event name to its ABI signature,
web3.eth.abi.encodeEventSignature('myEvent(uint256,bytes32)')

web3.eth.abi.encodeParameter('uint256', '2345675643');
web3.eth.abi.encodeParameters(['uint256','string'], ['2345675643', 'Hello!%']);

web3.eth.abi.decodeParameter('uint256', '0x00000...');
web3.eth.abi.decodeParameters(['string', 'uint256'], '0x0000000000..');

web3.utils

What are bloom filters?

A Bloom filter is a probabilistic, space-efficient data structure used for fast checks of set membership. That probably doesn't mean much to you yet, and so let's explore how bloom filters might be used. Imagine that we have some large set of data, and we want to be able to quickly test if some element is currently in that set.

The naive way of checking might be to query the set to see if our element is in there. That's probably fine if our data set is relatively small. Unfortunately, if our data set is really big, this search might take a while. Luckily, we have tricks to speed things up in the ethereum world! A bloom filter is one of these tricks.

The basic idea behind the Bloom filter is to hash each new element that goes into the data set, take certain bits from this hash, and then use those bits to fill in parts of a fixed-size bit array (e.g. set certain bits to 1). This bit array is called a bloom filter. Later, when we want to check if an element is in the set, we simply hash the element and check that the right bits are in the bloom filter.

// The BN.js library for calculating with big numbers
web3.utils.BN(mixed) //mixed - String|Number:

new web3.utils.BN(1234).toString() => "1234"
new web3.utils.BN('1234').add(new web3.utils.BN('1')).toString();

web3.utils.sha3(string)

web3.utils.isAddress(address)

// Will convert an upper or lowercase Ethereum address to a checksum address.
web3.utils.toChecksumAddress(address)

// Will auto convert any given value to HEX. Number strings will be interpreted as numbers.
// Text strings will be interpreted as UTF-8 strings
web3.utils.toHex(mixed)

// Will safely convert any given value
web3.utils.toBN(number)

web3.utils.hexToNumberString(hex)

web3.utils.hexToNumber('0xea') => 234

web3.utils.numberToHex('234') => '0xea'

web3.utils.hexToUtf8('0x49206861766520313030e282ac') => "I have 100C"

web3.utils.hexToAscii('0x4920686176652031303021') => "I have 100!"

web3.utils.utf8ToHex('I have 100C') => "0x49206861766520313030e282ac"

web3.utils.hexToBytes('0x000000ea') => [0, 0, 0, 234]

web3.utils.bytesToHex([72, 101, 108, 108, 111, 33, 36]) => "0x48656c6c6f2125"

web3.utils.toWei(number [, unit])
unit: wei, gwei, gwei, ether (default)
web3.utils.toWei('1', 'ether') => "1000000000000000000"

Returns: String|BN: If a string is given it returns a number string, otherwise a BN.js instance

web3.utils.fromWei('1', 'ether') => "0.000000000000000001" // always returns string number

web3.utils.padLeft('0x3456ff', 20) => "0x0000000000000003456ff"

web3.utils.padRight('Hello', 20, 'x') => "Helloxxxxxxxxxxxxxxx"