---

title: JavaScript API libraries description: An introduction to the JavaScript client libraries that let you interact with the blockchain from your application. lang: en

---

In order for a web app to interact with the Ethereum blockchain (i.e. read blockchain data and/or send transactions to the network), it must connect to an Ethereum node.

For this purpose, every Ethereum client implements the JSON-RPC specification, so there are a uniform set of methods that applications can rely on.

If you want to use JavaScript to connect with an Ethereum node, it's possible to use vanilla JavaScript but several convenience libraries exist within the ecosystem that make this much easier. With these libraries, developers can write intuitive, one-line methods to initialize JSON RPC requests (under the hood) that interact with Ethereum.

Please note that since The Merge, two connected pieces of Ethereum software - an execution client and a consensus client - are required to run a node. Please ensure your node includes both an execution and consensus client. If your node is not on your local machine (e.g. your node is running on an AWS instance) update the IP addresses in the tutorial accordingly. For more information please see our page on running a node.

Prerequisites {#prerequisites}

As well as understanding JavaScript, it might be helpful to understand the Ethereum stack and Ethereum clients.

Why use a library? {#why-use-a-library}

These libraries abstract away much of the complexity of interacting directly with an Ethereum node. They also provide utility functions (e.g. converting ETH to Gwei) so as a developer you can spend less time dealing with the intricacies of Ethereum clients and more time focused on the unique functionality of your application.

Library features {#library-features}

Connect to Ethereum nodes {#connect-to-ethereum-nodes}

Using providers, these libraries allow you to connect to Ethereum and read its data, whether that's over JSON-RPC, INFURA, Etherscan, Alchemy or MetaMask.

Ethers example

```js // A Web3Provider wraps a standard Web3 provider, which is // what MetaMask injects as window.ethereum into each page const provider = new ethers.providers.Web3Provider(window.ethereum)

// The MetaMask plugin also allows signing transactions to // send ether and pay to change state within the blockchain. // For this, we need the account signer... const signer = provider.getSigner() ```

**Web3js example**

```js var web3 = new Web3("http://localhost:8545") // or var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"))

// change provider web3.setProvider("ws://localhost:8546") // or web3.setProvider(new Web3.providers.WebsocketProvider("ws://localhost:8546"))

// Using the IPC provider in node.js var net = require("net") var web3 = new Web3("/Users/myuser/Library/Ethereum/geth.ipc", net) // mac os path // or var web3 = new Web3( new Web3.providers.IpcProvider("/Users/myuser/Library/Ethereum/geth.ipc", net) ) // mac os path // on windows the path is: "\\.\pipe\geth.ipc" // on linux the path is: "/users/myuser/.ethereum/geth.ipc" ```

Once set up you'll be able to query the blockchain for:

- block numbers
- gas estimates
- smart contract events
- network id
- and more...

### Wallet functionality {#wallet-functionality}

These libraries give you functionality to create wallets, manage keys and sign transactions.

Here's an examples from Ethers

```js // Create a wallet instance from a mnemonic... mnemonic = "announce room limb pattern dry unit scale effort smooth jazz weasel alcohol" walletMnemonic = Wallet.fromMnemonic(mnemonic)

// ...or from a private key walletPrivateKey = new Wallet(walletMnemonic.privateKey)

walletMnemonic.address === walletPrivateKey.address // true

// The address as a Promise per the Signer API walletMnemonic.getAddress() // { Promise: '0x71CB05EE1b1F506fF321Da3dac38f25c0c9ce6E1' }

// A Wallet address is also available synchronously walletMnemonic.address // '0x71CB05EE1b1F506fF321Da3dac38f25c0c9ce6E1'

// The internal cryptographic components walletMnemonic.privateKey // '0x1da6847600b0ee25e9ad9a52abbd786dd2502fa4005dd5af9310b7cc7a3b25db' walletMnemonic.publicKey // '0x04b9e72dfd423bcf95b3801ac93f4392be5ff22143f9980eb78b3a860c4843bfd04829ae61cdba4b3b1978ac5fc64f5cc2f4350e35a108a9c9a92a81200a60cd64'

// The wallet mnemonic walletMnemonic.mnemonic // { // locale: 'en', // path: 'm/44\'/60\'/0\'/0/0', // phrase: 'announce room limb pattern dry unit scale effort smooth jazz weasel alcohol' // }

// Note: A wallet created with a private key does not // have a mnemonic (the derivation prevents it) walletPrivateKey.mnemonic // null

// Signing a message walletMnemonic.signMessage("Hello World") // { Promise: '0x14280e5885a19f60e536de50097e96e3738c7acae4e9e62d67272d794b8127d31c03d9cd59781d4ee31fb4e1b893bd9b020ec67dfa65cfb51e2bdadbb1de26d91c' }

tx = { to: "0x8ba1f109551bD432803012645Ac136ddd64DBA72", value: utils.parseEther("1.0"), }

// Signing a transaction walletMnemonic.signTransaction(tx) // { Promise: '0xf865808080948ba1f109551bd432803012645ac136ddd64dba72880de0b6b3a7640000801ca0918e294306d177ab7bd664f5e141436563854ebe0a3e523b9690b4922bbb52b8a01181612cec9c431c42
}

// The connect method returns a new instance of the // Wallet connected to a provider wallet = walletMnemonic.connect(provider)

// Querying the network wallet.getBalance() // { Promise: { BigNumber: "42" } } wallet.getTransactionCount() // { Promise: 0 }

// Sending ether wallet.sendTransaction(tx) ```

Read the full docs

Once set up you'll be able to:

- create accounts
- send transactions
- sign transactions
```

- and more...

Interact with smart contract functions {#interact-with-smart-contract-functions}

JavaScript client libraries allow your application to call smart contract functions by reading the Application Binary Interface (ABI) of a compiled contract.

The ABI essentially explains the contract's functions in a JSON format and allows you to use it like a normal JavaScript object.

So the following Solidity contract:

```solidity contract Test { uint a; address d = 0x12345678901234567890123456789012;

```
function Test(uint testInt) { a = testInt;}

event Event(uint indexed b, bytes32 c);

event Event2(uint indexed b, bytes32 c);

function foo(uint b, bytes32 c) returns(address) {
    Event(b, c);
    return d;
}
```

} ```

Would result in the following JSON:

```
json [{ "type":"constructor", "payable":false, "stateMutability":"nonpayable" "inputs":[{"name":"testInt","type":"uint256"}], },{ "type":"function", "name":"foo", "constant":false,
"payable":false, "stateMutability":"nonpayable", "inputs":[{"name":"b","type":"uint256"}, {"name":"c","type":"bytes32"}], "outputs":[{"name":"","type":"address"}] },{
"type":"event", "name":"Event", "inputs":[{"indexed":true,"name":"b","type":"uint256"}, {"indexed":false,"name":"c","type":"bytes32"}], "anonymous":false },{ "type":"event",
"name":"Event2", "inputs":[{"indexed":true,"name":"b","type":"uint256"},{"indexed":false,"name":"c","type":"bytes32"}], "anonymous":false }]
~~~

This means you can:

- Send a transaction to the smart contract and execute its method
- Call to estimate the gas a method execution will take when executed in the EVM
- Deploy a contract
- And more...

Utility functions {#utility-functions}

Utility functions give you handy shortcuts that make building with Ethereum a little easier.

ETH values are in Wei by default. 1 ETH = 1,000,000,000,000,000,000 WEI – this means you're dealing with a lot of numbers! `web3.utils.toWei` converts ether to Wei for you.

And in ethers it looks like this:

```js // Get the balance of an account (by address or ENS name) balance = await provider.getBalance("ethers.eth") // { BigNumber: "2337132817842795605" }

// Often you will need to format the output for the user // which prefer to see values in ether (instead of wei) ethers.utils.formatEther(balance) // '2.337132817842795605' ```

- Web3js utility functions
- Ethers utility functions

## Available libraries {#available-libraries}

**Web3.js - *Ethereum JavaScript API.***

- Documentation
- GitHub

**Ethers.js - *Complete Ethereum wallet implementation and utilities in JavaScript and TypeScript.***

- Documentation
- GitHub

**The Graph - *A protocol for indexing Ethereum and IPFS data and querying it using GraphQL.***

- The Graph
- Graph Explorer
- Documentation
- GitHub
- Discord

**light.js - *A high-level reactive JS library optimized for light clients.***

- GitHub

**Web3-wrapper - *Typescript alternative to Web3.js.***

- Documentation
- GitHub

**Alchemyweb3 - *Wrapper around Web3.js with automatic retries and enhanced apis.***

- Documentation
- GitHub

**Alchemy NFT API - *API for fetching NFT data, including ownership, metadata attributes and more.***

- Documentation
- GitHub

**viem - *TypeScript Interface for Ethereum.***

- Documentation
- GitHub

## Further reading {#further-reading}

*Know of a community resource that helped you? Edit this page and add it!*
~~~

## Related topics {#related-topics}

- Nodes and clients
- Development frameworks

## Related tutorials {#related-tutorials}

- Set up Web3js to use the Ethereum blockchain in JavaScript– *Instructions for getting web3.js set up in your project.*
- Calling a smart contract from JavaScript – *Using the DAI token, see how to call contracts function using JavaScript.*
- Sending transactions using web3 and Alchemy– *Step by step walkthrough for sending transactions from the backend.*