

Tutorial 3

web3.js introduction (part 1)

Prepared by: Reza Parizi

In Tutorial 2 (election dapp), you used some code to connect to the blockchain (both in app.js (front-end) and for migration part). All that code you wrote used modules in web3.js. Let's learn more about web3.js.

What is web3.js?

Recall, there are a few different aspects to developing dapps with Ethereum:

Smart contract development - writing code that gets deployed to the blockchain with the Solidity programming language.

Developing websites or clients that interact with the blockchain - writing code that reads and writes data from the blockchain with smart contracts.

Web3.js enables you to fulfill the *second* responsibility: developing clients which allow you to interact with a local or remote Ethereum **node**. Remember, the Ethereum network is made up of nodes. Since every node has a copy of the blockchain's ledger, we could say web3.js is used to *interact with Ethereum blockchain*, using a HTTP or IPC connection. Web3.js is a collection of libraries that allow you to perform actions like send Ether from one account to another, read and write data from blockchain, call functions in smart contracts, create smart contracts, and so much more *programmatically*!

How to use web3.js?

1. Install web3.js:

```
npm install web3
```

[to check currently installed web3 version: `npm view web3 version`]

2. Create a web3 object/instance and set a provider:

```
const Web3 = require('web3');  
const web3 = new Web3('some-provider');
```

what is web3 provider? A provider links to a running node. You need to give your client/app a way to connect to the blockchain. Web3js support [3 different providers](#): HttpProvider, WebsocketProvider, and IpcProvider. Setting a Web3 Provider basically tells our code which node we should be talking to handle our reads and writes. It's kind of like setting the URL of the remote web server for your API calls in a traditional web app.

Ethereum node choices: There are a few ways you could connect to a node. For one, you could run your own Ethereum node with [Geth](#) or [Parity](#). But this requires you to download a lot of data from the blockchain and keep it in sync. Unless, you want to create your own private blockchain network, then it would be the only choice!

Mostly for convenience, you can use [Infura](#) to access an Ethereum node without having to run one yourself. Infura is a service that provides a remote Ethereum node for free. All you need to do is sign up and obtain an API key and the URL for the network you want to connect to.

Note: Metamask uses Infura's servers under the hood as a web3 provider.

Now that all of your dependencies are installed, you can start connecting to the blockchain (i.e. its node) and writing code with web3.js to read/write data. Web3.js has a rich [API](#). I highly recommend [browsing](#) through it.

First, you should fire up the Node console in your terminal like this:

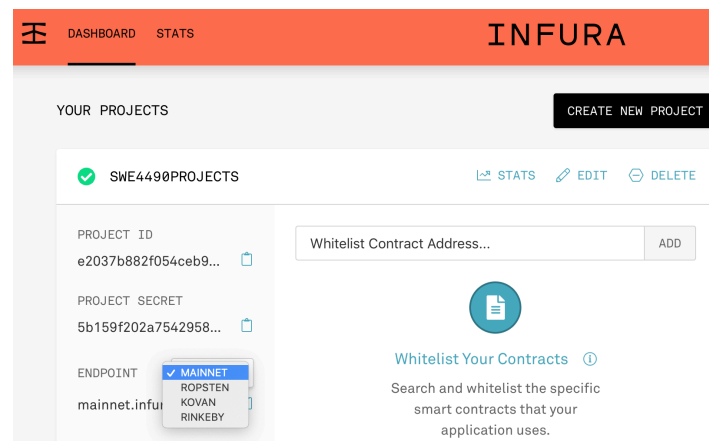
Node

Example 1: Checking an account balance (reading data from blockchain)—reading is free of gas

```
const Web3 = require('web3'); // 1. create a new Web3 connection/object
```

```
//2. set your provider, we use Infura. Simply copy the endpoint for your chosen network  
const rpcURL = "https://mainnet.infura.io/v3/YOUR-PROJECT-ID";
```

```
// NOTE: be sure to replace YOUR-PROJECT-ID with a Project ID from your Infura Dashboard. This url  
changes based on your network
```



```
const web3 = new Web3(rpcURL); // 3. instantiate a Web3 connection
```

```
//4. choose a random account from your wallet or on Etherscan  
const address = "0x5b7f69B1452068cD60810373D5f0875AeD967ACe";
```

```
//5. reading the balance of the account from blockchain  
var balance = web3.eth.getBalance(address, (err, wei) => {
```

```
    balance = web3.utils.fromWei(wei, 'ether')
  });
```

Let me explain this code (step 5). First, we use check the balance by calling `web3.eth.getBalance()`, which accepts a callback function/promise with two arguments, an error and the balance itself. We'll ignore the error argument for now, and reference the balance with the `wei` argument. Ethereum expresses balances in Wei, which is the smallest subdivision of Ether, kind of like a tiny penny. We can convert this balance to Ether with `web3.utils.fromWei(wei, 'ether')`.

```
//6. display the balance in Ether
console.log(balance);
```

Exercise: Change your network/web3 provider and see how the account's balance changes!

Example 2: Calling a smart's contract function (reading/writing data to blockchain)

Web3.js will need 2 things to talk to a contract: its **address** and its **ABI**.

Recall, when you compile your contract to deploy to Ethereum, the Solidity compiler will give you the ABI, so you'll need to copy and save this in addition to the contract's address. A smart contract ABI stands for "Abstract Binary Interface", and is a JSON array that describes how a specific smart contract works.

Once you have your contract's address and ABI, you can instantiate it in web3 as follows:

```
// Instantiate myContract
var myABI = blabla;
var myContractAddress = "blabla";

// this gives a JavaScript representation of the smart contract we want to interact with
var myContract = new web3.eth.Contract(myABI, myContractAddress);
```

Web3.js has TWO [methods](#) we will use to call functions on a contract: `call` and `send`

call is used for view and pure functions (querying to the blockchain). It won't create a transaction on the blockchain. Note: view and pure functions are read-only and don't change state on the blockchain. They also don't cost any gas, and the user won't be prompted to sign a transaction (with MetaMask or anything similar to that). In general, `call` can be very useful on public, payable, whatever function you want to “dry run.” `Call` is also useful in predicting if a function will revert without sending the transaction.

```
// call a function named myMethod in myContract contract with the parameter 123
myContract.methods.myMethod(123).call().then((result) => {
    console.log(result);
}).catch((err) => {
```

```

        console.log('error occurred: ${err}').red);
    });

```

send, will create a transaction and change data on the blockchain (along with contract's state). You'll need to use `send` for any functions that aren't view or pure. Note: sending a transaction requires a `from` address of who's calling the function (which becomes `msg.sender` in your Solidity code). We'll want this to be the user of our Dapp. Sending a transaction will require the user to pay gas, and will pop up their Metamask to prompt them to sign a transaction (if it is your provider). When we use Metamask as our web3 provider, this all happens automatically when we call `send()`, and we don't need to do any digital signature in our code. Pretty cool!

```

// send a transaction calling a function named myMethod with the parameter 34
myContract.methods.myMethod(34).send({from: 'address'}, (error, result) => {
    console.log(result);
}).catch((err) => {
    console.log('error occurred: ${err}').red);
});

```

Note: There is an alternative way to “send” using `web3js`, `ethereumjs-tx`. For more information refer to Tutorial 4.

Note that calls to smart contracts is **asynchronous**, like an API call to an external server. So Web3 returns a promise here. (If you're not familiar with [JavaScript promises](#)...time to do some additional homework!). Once the promise resolves (which means we got an answer back from the web3 provider), code continues with the `then` statement, which logs result to the console.

Exercise: let's write/use a simple contract to practice the above steps for calling functions in it.

Link to an existing deployed contract:

<https://ropsten.etherscan.io/address/0x632ed19b27cd1dacedac530c9042a216d88b6edf#code>

As an exercise, use OMGToken contract. You can find more details about this contract, including its ABI and address on [Etherscan](#). Use this smart contract to call its following two functions:

```

contract.methods.totalSupply().call().then((result) => {
    console.log(result);
}).catch((err) => {
    console.log('error occurred: ${err}').red);
});

```

```

contract.methods.name().call().then((result) => {
    console.log(result);
}).catch((err) => {
    console.log('error occurred: ${err}').red);
});

```

Note: `const contract = new web3.eth.Contract(abi, address);`

A few important notes:

Cryptography is complicated, so unless you're a security expert and you really know what you're doing, it's probably not a good idea to try to manage users' private keys yourself in your dapp's front-end. But luckily you don't need to—there are already services that handle this for you. The most popular of these is Metamask.

[Metamask](#) is a browser extension for Chrome and Firefox that lets users securely manage their Ethereum accounts and private keys, and use these accounts to interact with websites that are using Web3.js. (With metamask, your browser is web3 enabled, and you can interact with any website that communicates with the Ethereum blockchain!). And as a developer, if you want users to interact with your dapp through a website in their web browser, you'll definitely want to make it Metamask-compatible.

Metamask uses Infura's servers under the hood as a web3 provider, just like you did before—but it also gives the user the option to choose their own web3 provider. So by using Metamask's web3 provider, you're giving the user a choice, and it's one less thing you have to worry about in your app.