

Tutorial 4

web3.js introduction (part 2)

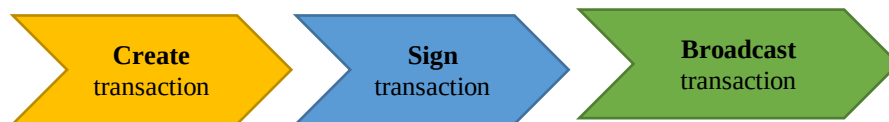
Ethereum transactions

Types of Transactions

The purpose of this part is to help you understand the fundamentals about how transactions work on the Ethereum blockchain. Broadly speaking there are THREE types of transactions supported on Ethereum:

1. Transferring of Ether from one account to another
2. Creating/deploying of a smart contract
3. Transacting with a smart contract (i.e., calling its functions)

With web3.js you can programmatically *create* transactions, *sign* a transaction and *broadcast* it to the network. You can get a greater understanding of these concepts by performing these actions with the web3.js library and observing how each step works through examples. Three main steps involved in any Ethereum transaction:



Installing dependency:

You'll need an additional JavaScript library to do this part, called [ethereumjs-tx](#). You can install this dependency from this command:

```
npm install ethereumjs-tx
```

Example 1: Transferring Ether from one account to another (type 1 transaction)

In this example, you're going to create a transaction that sends 0.1 Ether from account 1 to account 2. Let's get started.

Go to your node console, you'll first require the newly installed library like this:

```
var Tx = require('ethereumjs-tx');
```

Next, you'll set up a web3 connection like you did in the previous tutorial:

```
const Web3 = require('web3');  
const web3 = new Web3('https://ropsten.infura.io/xxx');
```

Notice, that we're using the Ropsten test network.

Create accounts (If needed)

We'll need two accounts and their private keys (assuming you have no prior accounts in any Ethereum wallet). You can actually create a new account from scratch with the following web3.js command:

```
web3.eth.accounts.create();
```

Note: run this command twice to create two accounts. See below screenshot.

```
[ksuParizi:~ rparizi1$ node
> var Tx = require('ethereumjs-tx');
undefined
> const Web3 = require('web3');
undefined
> const web3 = new Web3('https://ropsten.infura.io/v3/e2037b882f054ceb98697bd28c82e21b');
undefined
> web3.eth.accounts.create();
{ address: '0xD43df6aAA1FbAe325776c537cb92Ffd6D6f9046F',
  privateKey:
    '0xa7a502a3ad55412f4e510d2a65bd77dc80b53249f6760a705bfbda4129cd211a',
  signTransaction: [Function],
  sign: [Function],
  encrypt: [Function] }
> web3.eth.accounts.create();
{ address: '0x058886a135D56F5939115D988851DB0C8F9c4358',
  privateKey:
    '0x0c1c4553af57968e44553045a43f0073267beb32da64b04477a5fae1b38c62d2',
  signTransaction: [Function],
  sign: [Function],
  encrypt: [Function] }
> █
```

Once you have created both of these accounts (you can search for your accounts on [Etherscan](https://etherscan.io) to see their existence), make sure you load up account 1 with fake Ether from a faucet. You can obtain free Ether from a faucet on the Ropsten test network. Here are two faucets you can use: <http://faucet.ropsten.be:3001/> <https://faucet.metamask.io/> (have to import your new account to Metamask first)

Now, you'll assign them to variables in your script like this:

```
const account1 = '0xD43df6aAA1FbAe325776c537cb92Ffd6D6f9046F';
const account2 = '0x058886a135D56F5939115D988851DB0C8F9c4358';
```

[Important]: It's bad practice to expose private keys in your code (we don't hard code them into our file). What if you accidentally committed them to source in a real project? Someone could steal your Ether! Here is what I suggest to do: save your private keys to your environment then read those private keys from your environment and store them to variables/constants. We can do like this:

```
const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex');
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex');
```

Note, In order to sign transactions with the private keys, we must convert *raw* private keys to a string of binary data with a Buffer (a globally available module in NodeJS) **without** '0x' prefix.

Example:

```
const privateKey1 =
Buffer.from('0xa7a502a3ad55412f4e510d2a65bd77dc80b53249f6760a705bfbda4129cd211a', 'hex');
```

1. Create the transaction

Now, build the transaction and set its parameters like this:

```
var txObject;
web3.eth.getTransactionCount(account1, (err, txCount) => {
  txObject = {
    nonce:      web3.utils.toHex(txCount),
    to:         account2,
    value:      web3.utils.toHex(web3.utils.toWei('0.01', 'ether')),
    gasLimit:   web3.utils.toHex(21000),
    gasPrice:   web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }
});
```

Let me explain this code. We're building an object that has all the values/parameters needed to generate a raw transaction. Let's break down each of these values:

- **nonce** - It's simply the current transaction count of an account (known as **account nonce**), which is an increasing numeric value that is used to uniquely identify a transaction. You should assign the value of this variable momentarily, using `web3.eth.getTransactionCount("account", )` function to get the numbers of transactions sent from this address up to now. You also must convert this value to hexadecimal. We can do this with utility `web3.utils.toHex(..)`. nonce for a transaction is used to maintain the order in which transactions are processed. A nonce can only be used once and until a transaction is mined. So if you send a transaction with a nonce which is not one more than the previous transaction from that same account then that transaction will be rejected. This prevents replay attacks where a transaction sending e.g. 20 ethers from A to B can be replayed by B over and over to continually drain A's balance.

Note: nonce which you see here is not to be confused with the nonce that is used in the mining process (called **Proof of work nonce**), which is a random value in a block that is used get the proof of work satisfied depending on the difficulty at the time.

- **to** - the account we're sending Ether to.
- **value** - the amount of Ether we want to send. This value must be expressed in Wei and converted to hexadecimal. You can convert the value to wei with the Web3.js utility `web3.utils.toWei()`.

- Note*, that there is no ‘from’ field in the transaction object. That will be inferred whenever we sign this transaction with ‘account1’s’ private key.

```
const tx = new Tx(txObject);
tx.sign(privateKey1);

const serializedTx = tx.serialize();
const raw = '0x' + serializedTx.toString('hex');
```

```
web3.eth.sendSignedTransaction(raw, (err, txHash) => {
  console.log('txHash:', txHash)
});
```

[illegible]

```
var balance = web3.eth.getBalance(account1, (err, wei) => {
  balance = web3.utils.fromWei(wei, 'ether');
  console.log(balance);
});
```

5

```
    console.log(balance);  
  });
```

Example 2: Deploying a Smart Contract with Web3.js (type 2 transaction)

There are multiple ways you can deploy smart contracts to the Ethereum blockchain. Here, we want to learn how to do it using web3.js itself (hard coding).

The general steps are the same, it always consists of three same basic steps (build, sign, and broadcast the transaction). Deploying a smart contract actually looks a lot like sending Ether from one account to another (example 1). The only difference is the transaction parameters:

```
var txObject;  
web3.eth.getTransactionCount(creatorAccount, (err, txCount) => {  
  txObject = {  
    nonce:    web3.utils.toHex(txCount),  
    gasLimit: web3.utils.toHex(4700000),  
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),  
    data: ' ' //your data value goes here  
  }  
});
```

- **value** - this parameter is absent in this type of transaction because we aren't sending any Ether in this transaction (*unless* you wish to deposit some ethers in the smart contract-- assumes zero if not provided).
- **to** - this parameter is absent because we aren't sending this transaction to a particular account. Instead we're sending it to the entire network because we're deploying a smart contract!
- **data** - this will be the bytecode of the compiled smart contract that we want to deploy. In order to obtain this value, you first need a smart contract, and then you need to compile it (Using Remix perhaps): Remix> compile your smart contract> 'Details' button> under 'WEB3DEPLOY' locate data: and copy its value from there.

You are welcome to use any smart contract you like. Sign and broadcast your transaction to the network (like example 1). In the end, use txHash to search for your newly deployed contract on [Etherscan](#)). From there, you can publish your source code as we learned in Tutorial 1.

Example 3: Calling Smart Contract Functions with Web3.js (type 3 transaction)

The general steps are the same (like previous examples). The only difference is the transaction parameters:

```
var txObject;  
web3.eth.getTransactionCount(callerAccount, (err, txCount) => {  
  txObject = {
```

```

    nonce: web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(4700000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    to: ' ',
    data: contractObject.methods.functionName(parameters).encodeABI()
  }
});

```

- **to** - this parameter will be the address of the deployed contract.
- **data** - this will be the hexadecimal representation of the function we want to call on the smart contract with its parameter arguments. You can generate it using this code:
`contractObject.methods.functionName(parameters).encodeABI()`

As an exercise, try to use this deployed contract to call its `set` function with input parameter of 69: <https://ropsten.etherscan.io/address/0x632ed19b27cd1dacedac530c9042a216d88b6edf#code>

Your code should look like this:

```

//include ethereumjs-tx library and set up a web3js connection for your network
var Tx = require('ethereumjs-tx');
const Web3 = require('web3');
const web3 = new Web3('https://ropsten.infura.io/v3/xxx');

// account (the caller)
const account1 = '0xD43df6aAA1FbAe325776c537cb92Ffd6D6f9046F';
const privateKey1 = Buffer.from('xxxxxx', 'hex');

// setup contract info
var myABI = blabla; //read it from Etherscan
var myContractAddress = "0x632ed19b27cd1DACEdAc530c9042A216D88b6EDf";
var myContract = new web3.eth.Contract(myABI, myContractAddress);

//1. create the transaction
var txObject;
web3.eth.getTransactionCount(account1, (err, txCount) => {
  txObject = {
    nonce: web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(4700000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    to: myContractAddress,
    data: myContract.methods.set(69).encodeABI()
  }
});

//2. sign the transaction
const tx = new Tx(txObject);
tx.sign(privateKey1);

const serializedTx = tx.serialize();
const raw = '0x' + serializedTx.toString('hex');

//3. broadcast the signed transaction to the network
web3.eth.sendSignedTransaction(raw, (err, txHash) => {
  console.log('txHash:', txHash)
});

```

```
});  
  
// let's check to see the change by calling get function of the contract  
myContract.methods.get().call().then((result) => {  
    console.log(result);  
}).catch((err) => {  
    console.log('error occurred: ${err}'.red);  
});
```