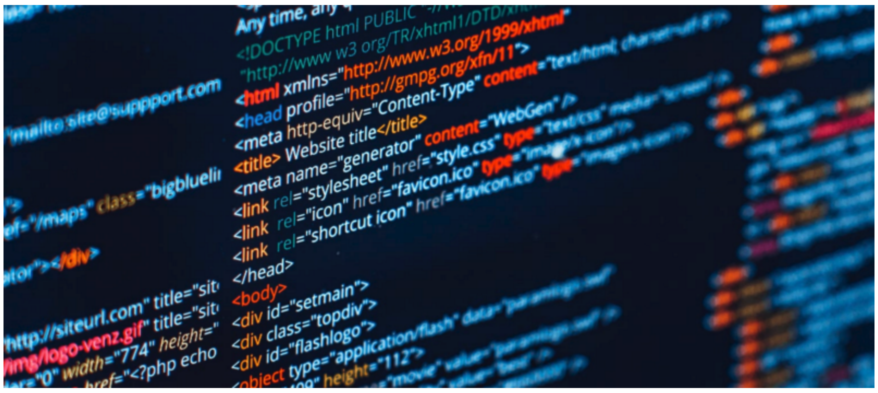
Alex Miller Follow
Ethereum developer at ConsenSys, co-founder of https://gridplus.io
Apr 11 · 12 min read



Getting Started as an Ethereum Web Developer

UPDATE: *I have created a repo for you to play around with that showcases most of the stuff covered in this article.*

I routinely build web applications that use Ethereum and I think I take for granted the amazing toolset that I use every day. Our ecosystem is growing rapidly and I think a lot of newcomers are feeling overwhelmed. Ethereum is an amazing technology, but it is also nascent and there simply hasn't been enough time for expertise to sufficiently permeate. I want people to know that Ethereum development is actually very compatible with modern web developer workflows—**it's relatively easy to integrate Ethereum functionality into any web application, and you can start today**.

Because I fancy myself a champion of Ethereum on a mission to show mainstream developers the light, I have decided to put a bunch of scattered knowledge into one place (not very decentralized, I know). You will of course need to consult the proper documentation at each step, but my hope is that this article will show you how everything, more or less, fits together.

If you're ready to learn, please let me be your spirit guide. Come join the Ethereum ecosystem and help us conquer the world.

Step 1: Get a blockchain

There are lots of clients to choose from, but I suggest not yet worrying about geth vs parity vs pyethapp (the up and coming python client— represent!). For everyone who just wants a freaking blockchain so they can start building stuff (e.g. you), I suggest testrpc for all your development needs. Once you have it installed, you can start it with:

```
testrpc
```

Congratulations, you now have a blockchain. Note that by default testrpc does not mine blocks, but the `-b` flag allows you to specify a block interval (e.g. 1 second). I like this configuration for a bunch of reasons I won't get into, but just keep in mind that it is available.

Step 2: Talk to the blockchain

Once you have your blockchain spinning, you need a way to talk to it. You have probably already downloaded web3.js. If you haven't, you must be *really* new. Well, go ahead and make sure you have web3 installed, then open up a `config.js` file and put this in it:

```
var web3 = require('web3');
var web3_provider = 'http://localhost:8545';
var _web3 = new web3();
_web3.setProvider(new
web3.providers.HttpProvider(web3_provider));
exports.web3 = _web3;
```

Any time you want to talk to the blockchain on your backend server, just do this:

```
var config = require('./config.js');

config.web3.eth.X
```

The `X` (i.e. whatever web3 API function you want) can be found here.

Step 3: Write some smart contracts

I'll save you some time here: you're going to be using solidity to write smart contracts. If you think smart contracting is scary, don't. For many applications, it's actually really easy as long as you follow one rule: **keep your contracts simple**.

There are 2 reasons you want to **always always always keep your contracts as absolutely stupid simple as is humanly possible**:

1. Every compute/storage operation costs gas, which equals ether, which equals money. We're talking the difference between paying $0.05 and $1.50 to call your contract. The point of Ethereum is not to replace your database (at least not in my opinion), so keep the logic short and the storage minimal.
2. More complexity = more places for things to go wrong. This is bad when your code is responsible for people's money and can't be rolled back. Please take a minute to let that last sentence sink in.

Okay, simple contracts—got it. Let's move on.

Step 4: Deploy those smart contracts

TRUFFLE

If you haven't heard of truffle, you should definitely check that out now. I like to manage my tester contracts in a truffle directory. The neat thing about this is that you can easily work it into your testing framework. Consider this script in package.json:

```
"scripts": {
  "test": "cd truffle && truffle deploy && truffle test
./myTruffleTest.js && cd .. && npm run myOtherTests"
}
```

What this is doing is 1. deploying your contracts, 2. running your truffle test, 3. running your regular tests—all in the same script!

Note that your truffle tests are "special" in that they inject a bunch of cool blockchainy stuff into your testing scope. There are a variety of ways to pass this information to the rest of your test suite. I personally use a truffle test to save the contract addresses into a config file and then import that config into my regular mocha tests. *As long as I have the proper addresses, I can interact with my contracts in any test via web3.js.* Anyway, you'll figure out what works best for you.

Back to the main show. You can deploy your smart contract(s) by going to your truffle directory and typing:

```
truffle deploy
```

Note that testrpc must be running in another window!

This will print the address of your freshly deployed contract, which you will need later. As I mentioned, you can always save this address programmatically in a truffle test, but for now you can just copy and paste it into your `config.js` file:

```
exports.contract_addr =
'0xe73e8e0a4442e140aea87a4b150ef07b82492500'
```

Step 5: Make a smart contract call

Now that we have a contract, we need to call it. Okay this one is going to seem janky—we'll be calling contracts with pure hex strings. Sure there are libraries to make this easier, but when it comes to contract calls, I kick it old school. And remember, I am your sensei.

The first thing to note is that **everything must be in hex** (see appendix for more details). Numbers, strings, etc. The second thing to note is that **words in Ethereum are 256 bits**. This means you need to left-pad everything with zeros to 64 characters. The third thing to note is that **types must be declared canonically** in the function definition.

Okay this is getting really neckbeardy. Let's go through an example:

```
function add(uint x, uint y) public constant returns (uint)
{
  return x + y;
}
```

Let's say you want to add `1` and `2`. Here's how you call this function:

1: Take the first 4 bytes of the keccak 256 hash of your tightly packed, canonical function definition.

Say what? Well, I didn't make this up, but you can just type your function declaration into this website and take the first 8 characters. What do I mean by canonical? Well, in Ethereum there are canonical types and shorthand types (e.g. `uint256` is `uint`'s canonical type). I actually don't know where they're all defined, but check out the Ethereum ABI definition for examples as well as this post.

Anyway, this is what our declaration looks like:

```
add(uint256,uint256)
```

Which returns the keccak256 hash:

```
771602f7f25ce61b0d4f2430f7e4789bfd9e6e4029613fda01b7f2c89fbf
44ad
```

Of which the first 4 bytes (8 characters) are:

```
771602f7
```

2: Pad your parameters to 256 bits

This one's a little easier to get a grasp on:

x=1 is:

```
000000000000000000000000000000000000000000000000000000000000
0001
```

y=2 is:

```
000000000000000000000000000000000000000000000000000000000000
0002
```

And together they are:

```
000000000000000000000000000000000000000000000000000000000000
000100000000000000000000000000000000000000000000000000000000
00000002
```

3: Pack everything together and add a `0x` *prefix*

Pretty self explanatory:

```
0x771602f700000000000000000000000000000000000000000000000000
000000000000010000000000000000000000000000000000000000000000
000000000000000002
```

—

Now that we have our payload, we can call the contract with web3:

```
var config = require('./config.js');
var call =
'0x771602f70000000000000000000000000000000000000000000000000
000000000000001000000000000000000000000000000000000000000000
0000000000000000002'

var to = config.contract_addr;

var res = config.web3.eth.call({ to: to, data: call });
```

After that, you should get back `3` for `res`. Actually, you'll get a `BigNumber` object:

```
res.toString()
>'3'
```

You should probably read this to learn more about why you should use `BigNumber` s throughout your app.

Okay fine, you can use the library I was talking about earlier.

Wait, we're not done yet! I have only just shown you how to **call** a contract. But what if you want to write to it (i.e. update state)? The above will not work! You need to sign a transaction with your private key, but before that, you need some ether.

Step 6: Setup your account

Let's go back to truffle. In our test, we need to add something like this:

```
var keys = require(`${process.cwd()}/../test/keys.json`);

it('Should send me some ether.', function() {
  assert.notEqual(keys.me.addr, null);

  var eth = 1*Math.pow(10, 18);
  var sendObj = {
    from: accounts[0],
    value: eth,
    to: keys.me.addr
  }

  Promise.resolve(web3.eth.sendTransaction(sendObj))
  .then(function(txHash) {
    assert.notEqual(txHash, null);
    return web3.eth.getBalance(keys.me.addr)
  })
  .then(function(balance) {
    assert.notEqual(balance.toNumber(), 0);
  })
})
```

Important note: we are actually sending `1 ether` *which is the same as* `10^18 wei`*. We **always** make calls/transactions with **wei** values.*

Now, I'm skipping a step here. You need to first get an Ethereum account, which is derived from a private/public keypair that you generate. I like using eth-lightwallet for key management on the backend. If you want more details/instructions on that, they are in the appendix.

To keep things simple, let's just pretend I have this variable hardcoded in my ever growing `config.js` :

```
exports.me = {
  addr: "0x29f2f6405e6a307baded0b0672745691358e3ee6",
  pkey:
"8c2bcfce3d9c4f215fcae9b215eb7c95831da0219ebfe0bb909eb951c31
34515"
}
```

*Obligatory reminder: **never** share your private key, upload it to github, or publish it on Medium if there is or will ever be any money on it.*

Back to the test, you can see that ether is being moved from `accounts[0]` , which by default has a bunch of ether, to `me.addr` , which is in your config file.

Step 7: Transacting with your smart contracts

Now that your account has some ether, it's time to spend it. There are 3 ways to spend ether:

1. Sending it to another address as `value`
2. Calling a contract function that updates the state of the network, which requires gas to incentivize the miner to process your update.

3. Call a contract that updates state, but also accepts ether as payment (FYI there is a `payable` modifier in solidity)— `value` will be sent and you will also have to pay for gas.

What we're going to do next falls into category #2. Suppose we have the following function that keeps track of a user's balance of something:

```
function addUserBalance(uint balance)
public returns (bool) {
  if (!accounts[msg.sender]) { throw; }
  if (accounts[msg.sender].balance + balance <
accounts[msg.sender].balance) { throw; }
  accounts[msg.sender].balance += balance;
  return true;
}
```

Note the second `if` statement, which is necessary because adding and subtracting in solidity can lead to numerical overflow and underflow—be careful! Also note the undeclared `msg` object living in your function scope. This has all sorts of neat stuff you can reference in your function.

When we call this function by sending a transaction, we are asking to update the global state of the network to say the following:

> *The balance of* `msg.sender` *'s account, within the scope of this contract, has been increased by* `balance` .

We don't have the power to update state ourselves, so we need a miner to do it for us. We pay him or her for this service with `gas` , which translates to ether.

To appropriately call this function, we need to use ABI again:

```
addUserBalance(uint256) --> 22526328 -->
0x22526328000000000000000000000000000000000000000000000000
00000000000001
```

And we use this data to form an **unsigned** transaction:

```
var data =
'0x225263280000000000000000000000000000000000000000000000000000
000000000000001';
var nonce =
config.web3.eth.getTransactionCount(keys.me.addr);
var gasPrice = 20 * Math.pow(10, 9);
var gasLimit = 100000;

var txn = {
  from: config.me.addr,
  to: config.contract_address,
  gas: `0x${gasLimit.toString(16)}`,
  gasPrice: `0x${gasPrice.toString(16)}`,
  data: data,
  nonce: `0x${nonce.toString(16)}`,
  value: '0x0'
}
```

As mentioned above, `gas` is required to make a transaction (i.e. update the state). `gas * gasPrice` is the amount of wei the miner can possibly spend to execute your transaction. If the operations cost more than what you provided, the transaction will not update the state and the miner will keep all your gas money. If less than the gas provided is used, you are refunded the remainder.

If we submit this object to the network, it will fail because there has been no proof that **I** am actually authorizing this transaction. Who knows, some stranger could be updating my balance to 1 billion (although it's unclear why anyone would do that).

Anyway, what I need to do is **sign** the transaction with my private key. Remember that old thing hanging out in your config file that I told you not to share with anyone? Do this with it:

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

var privateKey = Buffer.from(config.me.pkey, 'hex')
var tx = new Tx(txn);
tx.sign(privateKey);
var serializedTx = tx.serialize();
```

Here we are using one of my favorite libraries to sign a transaction object given your private key. This should return something like the following:

```
0xf8aa808504a817c800830f424094a0f68379088f9aee95ba5c9d178693
b874c4cd6880b844a9059cbb000000000000000000000000053b2188b0b1
00e68299708864e2ccecb62cdf0d00000000000000000000000000000000
0000000000000000000000746a5288001ca01f683f083c2d7c741a1218ef
c0144adc1749125a9ca53134b06353a8e4ef72afa07c50fb59647ff8b889
5b75795b0f51de745fa5987b985f7d1025eb346755bca0
```

Now, finally, we can submit this to the blockchain via web3. It will return a transaction hash that is simply a hash of the transaction provided (*this is, very importantly, **not** proof that the transaction was successful!*)

```
var txHash = config.web3.eth.sendRawTransaction(raw_txn);
```

Which looks something like this:

```
0xac8914ecb06b333a9e655a85a0cd0cccddb8ac627098e7c40877d27a13
0a7293
```

Now there's one last step, which is strictly speaking optional, but important to verify that your transaction has been accepted and processed: getting your transaction receipt.

```
var txReceipt =
config.web3.eth.getTransactionReceipt(txHash);
```

If this returns `null` your transaction was not picked up (perhaps you signed with the wrong private key?). If it's not `null` , there may still be various other clues that your transaction failed that I won't get into now.

Okay, just one clue—if your `gasUsed` is equal to the total `gas` sent, it means your function call failed. This means either 1. you didn't provide enough gas and/or 2. your contract encountered a `throw` .

And that's a wrap!

Conclusion

I know, that was a lot of stuff.

If you're feeling overwhelemed, I suggest you take it slow and use this article as a reference. You will probably need to spend a lot of time at each step reading documentation and playing around.

That said, what I've described above is 80% of the battle. Once you've mastered these things, I would personally consider you a capable Ethereum developer.

If you're interested, start tinkering! The tools are getting better by the day and it's never been easier to jump in. Welcome aboard.

—

If you liked this article, follow me on twitter or come join the community. Also check out my company, ConsenSys, because we do awesome stuff as well as my project, Grid+, which is using Ethereum to change the future of energy. If you want to join our mailing list, you can subscribe here:

This embedded content is from a site that does not comply with the Do Not Track (DNT) setting now enabled on your browser.

Please note, if you click through and view it anyway, you may be tracked by the website hosting the embed.

—

Appendix:

Hex strings:

I realized I didn't choose the best example for hexadecimal numbers (since `1` is the same value in both hex and base10 notation). If you were to instead call our `add` function with this as one of the parameters:

```
0000000000000000000000000000000000000000000000000000000000000
0100
```

You would actually be adding `256`. So probably what's good practice for any function call is to do the following:

```
var _x = 100;
var x = `${x}.toString(16)`;
```

This will ensure you are always calling with hex values.

Keys and profiles

If you're stuck on how to generate keys, you can go ahead and look at this file I use in my apps:

```
/**
 * Generate a test keypair
 */

var keystore = require('eth-lightwallet').keystore;
var Promise = require('bluebird').Promise;
var crypto = require('crypto');
var fs = Promise.promisifyAll(require('fs'));
var jsonfile = Promise.promisifyAll(require('jsonfile'));

var KEYSTORE_DIRECTORY = `${process.cwd()}/test`;
var name = process.argv[2];
var password = "test";

/**
  * Run the generator
  */
createKeystore(password)
.then(function(ks) {
  return saveProfile(name, ks.keystore, ks.privateKey,
ks.address); })
.then(function(saved) { console.log('saved', saved); })
.catch(function(error) { console.log(error); });


/**
  * Utility Functions
  */
function createKeystore(_password) {
  return new Promise(function(resolve, reject) {
    var password = Buffer(_password).toString('hex');
    keystore.createVault({ password: password },
function(error, ks) {
      if (error) { reject(error); }
      ks.keyFromPassword(password, function(error, dKey) {
        if (error) { reject(error); }
        ks.generateNewAddress(dKey, 1);
        var address = `0x${ks.getAddresses()[0]}`;
        var privateKey = ks.exportPrivateKey(address, dKey);
        var keystore = JSON.parse(ks.serialize());
        resolve({ address, privateKey, keystore });
      });
    });
  });
}
```

```
function saveProfile(name, keystore, privateKey, address) {
  return new Promise((resolve, reject) => {

jsonfile.readFileAsync(`${KEYSTORE_DIRECTORY}/keys.json`,
{throws: false})
    .then(function(PROFILES) {
      var profiles = PROFILES || {};
      profiles[`${name}`] = {
        keystore,
        privateKey,
        address
      };
      console.log('profiles', profiles)
      return profiles;
    })
    .then(function(_profiles) {
      return
jsonfile.writeFileAsync(`${KEYSTORE_DIRECTORY}/keys.json`,
_profiles, {spaces: 2});
    })
    .then(function() { resolve(true); })
    .catch(function(error) { reject(error); });
  })
}
```

This will save a keystore that you name in a file called `test/keys.json`. You can run it with plain old `node keygen.js <account_name>` and you should be good. I use this for test accounts, but you can reuse the functions if you need to e.g. automatically generate accounts.

Hacker Noon is how hackers start their afternoons. We're a part of the @AMI family. We are now accepting submissions and happy to discuss

> *advertising & sponsorship opportunities.*

> *If you enjoyed this story, we recommend reading our latest tech stories and trending tech stories. Until next time, don't take the realities of the world for granted!*