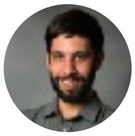
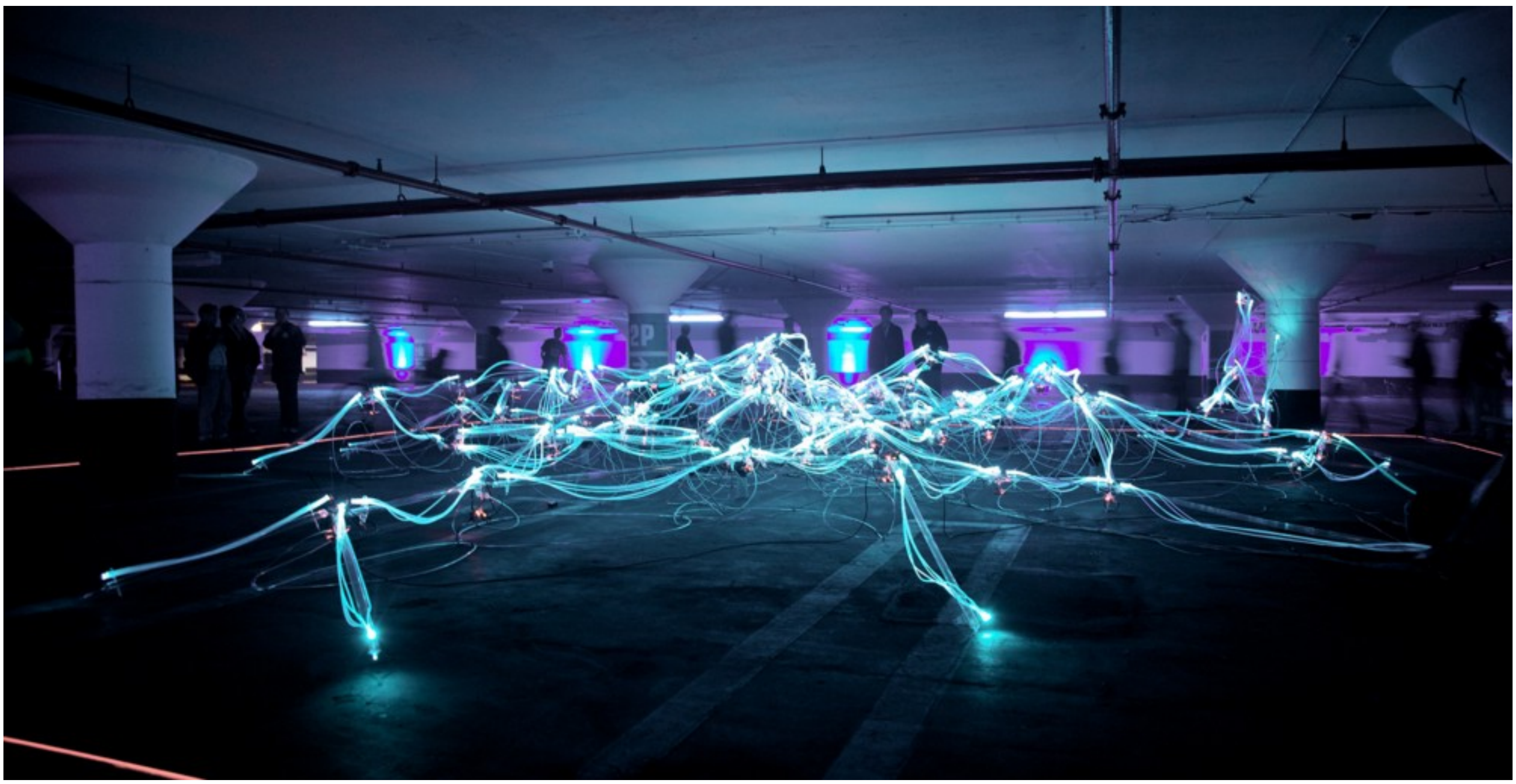


Júlio Santos Follow
Experienced CTO now helping businesses build and launch technology products. Founder and Managing Partner at Life on Mars: https://lifeonmars.pt
Nov 28, 2017 · 17 min read

Full-stack smart contract development

Writing, testing and deploying an Ethereum smart contract and its web interface



This weekend I spent some time with my team looking into tooling and deployments particular to the Ethereum blockchain, and put together a little experiment: Forever on the Chain.

It's the equivalent of a digital tattoo: a smart contract that anyone can use for free (minus transaction costs) to leave a permanent message onto the Ethereum blockchain. This message is stored in the blockchain forever, etched into thousands upon thousands of computers, unchangeable and immortal.

Even though permanence is one of the core blockchain concepts, this still worried some folks, so I also wrote a non-technical post about the tool and some of the implications of blockchain technology.

What follows is a smart contract tutorial. I'll walk you through the creation and testing of the critical path of this tool.

This assumes you have an understanding of some Ethereum core concepts such as smart contracts, transactions, and gas. If you don't, here's an explainer to get you started.

How does Ethereum work, anyway?

Introduction

medium.com

The smart contract

This smart contract is as simple as they come. Its only functionality is to log a message into the blockchain. This is achieved through the use of Events, as explained below.

Code walkthrough

The smart contract is written in Solidity. This is a statically typed language to write Ethereum smart contracts. From the documentation:

> *Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript and it is designed to target the Ethereum Virtual Machine (EVM).*

Here's what the implementation of our contract looks like.

```
pragma solidity 0.4.18;

/**
 * @title Recorder – record a message into the blockchain
 * @author Life on Mars – https://lifeonmars.pt
 */
contract Recorder{
  event Record(
    address _from,
    string _message
  );

  /**
   * @notice Sends the contract a message
   * to record into the blockchain
   * @param message  message to record
   */
  function record(string message) {
    Record(msg.sender, message);
  }
}
```

Let's start by walking slowly through this code.

```
pragma solidity 0.4.18;
```

As per the documentation:

> *Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible changes.*

This line ensures that your source file won't be compiled by a compiler with a version different from 0.4.18.

It's also possible to use less restrictive pragmas, for example through a caret range. Take `^0.4.18` . In this case, your source will be compiled by any compiler later than `0.4.18` , and earlier than `0.5.0.` See the semver documentation for npm for more information.

```
contract Recorder {}
```

`contract` is, as the name implies, the keyword one uses to define a contract. A contract is defined with a name written in PascalCase.

```
event Record(
  address _from,
  string _message
);
```

Here, we're defining an event. Events allow you write access to the Ethereum logging facilities. When called, the arguments they were invoked with get stored in the transaction's log, which is a special data structure in the blockchain [docs]. These transaction logs can be used for storing information.

Compared to using contract storage (writing to a variable in a contract), using logs is much cheaper with regards to gas costs. However, this comes with a trade-off: contracts aren't able to read from log data (see this great post by Consensys for more details).

Since, for our use case, we don't need contracts to read from these logs, we're using events instead of storing an array of strings.

```
function record(string message) {
  Record(msg.sender, message);
}
```

The `record` method is as simple as they come. It takes a `string` argument named `message`, and all it does is call the `Record` event with two parameters:

- `msg.sender` holds the address of the account which invoked the `record` function [docs];
- `message` is just the argument `record` was invoked with.

Interaction with this particular smart contract is possible in only one way: sending a transaction at it, along with a `message` parameter, which invokes the `record` method. When an account `A` invokes the `record` method, the `Record` event is called, which causes the tuple `{A's address, message}` to be stored in the respective transaction's log.

Compilation

Solidity code can be compiled using `solc`, the solidity commandline compiler. We will see how to use Truffle for compilation and deployment later on, but bear with me for now. **Install solc if you want to follow along.**

Running `solc` allows us to output the binary representation of the contract. This is used for deploying the contract (again, Truffle will handle this for us later). Here's what this contract's binary output looks like (you can see it yourself with `solc --bin Recorder.sol`)


```
60606040523415610 00f57600080fd5b6101a28061001e6000396000f30
```

```
00191505b50935050505060405180910390a1505600a165627a7a723058
201b9ef174b2b5557ad31400e4eb1cd02dd02b8244eefbc97e9b75b0107
2cc706e0029
```


Pretty exciting stuff.

As per the official documentation:

> *The encoding is not self describing and thus requires a schema in order to decode.*

What this means is that, if we are to be able to understand and interact with a contract, we need to know its ABI specification. An ABI (Application Binary Interface) is the interface specification of a program.

Here's what that specification looks like for this contract (you can see it yourself by `solc --abi Recorder.sol`).


```
[
  {
    "constant": false,
    "inputs": [
      {
        "name": "message",
        "type": "string"
      }
    ],
    "name": "record",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": false,
        "name": "_from",
        "type": "address"
      },
      {
        "indexed": false,
        "name": "_message",
        "type": "string"
```

```
      }
    ],
    "name": "Record",
    "type": "event"
  }
]
```

This ABI spec will become relevant later on.

Natspec

The compiler can also produce documentation based on your use of Ethereum's Natural Specification Format, or Natspec. This can be used by UIs to display additional information to users and/or developers. This contract has such comments for itself and also for its `record` method.

Here's the compile output.

```
"devdoc": {
  "author": "Life on Mars — https://lifeonmars.pt",
  "methods": {
    "record(string)": {
      "params": {
        "message": " — message to record"
      }
    }
  },
  "title": "Recorder — record a message into the
blockchain"
},
"userdoc": {
  "methods": {
    "record(string)": {
      "notice": "Sends the contract a message to record
into the blockchain"
    }
  }
}
```

. . .

I wanted to show you these internals, but we'll be using Truffle from now on. Truffle is a framework for Ethereum, with a few niceties that'll make our life easier. In our case, it'll help with compilation and testing.

You'll need Truffle in order to follow along the next sections, so please follow the instructions to install it.

Deploying and testing the contract on a local network

In order to test and deploy the contract, we'll need an Ethereum client running. We'll get started with testrpc, which simulates the behaviour of a real client (such as geth, which we'll look into later on), but it's much faster. This speed contributes to faster development and testing cycles.

As per the instructions, install it with `npm install -g ethereumjs-testrpc` and run it with `testrpc`.

Now, create a new `recorder` directory, `cd` into it, and run `truffle init`. We'll need to configure Truffle to connect to `testrpc`. Edit the `truffle.js` file as follows:

```
ℹ️ truffle.js
-------------


module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "*",
      gas: 4712387,
    }
  }
};
```

Next, create a `contracts/Recorder.sol` file where you'll store the contract's source code, shown above.

Then create the `migrations/2_deploy_contracts.js` as such:

```
ℹ️ migrations/2_deploy_contracts.js
-----------------------------------

var Recorder = artifacts.require("./Recorder.sol");

module.exports = function(deployer) {
  deployer.deploy(Recorder);
};
```

Now we're ready to open the `truffle console`, where we'll get Truffle to deploy the Recorder contract into the network through `migrate --reset`. Truffle will make sure to compile your contract automatically for you.

(By the way, `--reset` tells truffle to run all the migrations from the beginning. For the purposes of this post, we're not interested in using migrations for contract versioning, so we'll always use it to start the process anew.)

Once the contract is deployed, we're ready to use it! We'll store a couple of records and then proceed to query them.

We can call methods on contracts by acquiring a contract reference and calling the method directly on it:

```
ℹ️ truffle console
-----------------

truffle(development)> recorder =
Recorder.at(Recorder.address)
truffle(development)> recorder.record("first message");
truffle(development)> recorder.record("second message");
```

Note the `testrpc` output. Among other things, it'll tell you how much gas was used up to run this transaction.

```
ℹ️ testrpc output
-----------------

eth_sendTransaction

Transaction:
0x118de578d3bcbd9474099e5385b0a883ad18eaa99319f710f641db1cc
6b2e8fe
  Gas usage: 24577
  Block Number: 14
  Block Time: Fri Nov 10 2017 20:06:42 GMT+0000 (WET)

eth_getTransactionReceipt
```

In order to read what was written earlier, we'll need to run the following.

```
ℹ️ truffle console
------------------

truffle(development)> recorder
  .Record({}, {fromBlock: 0, toBlock: "latest"})
  .get((error,result) => (
    console.log(result.map(
      (result) => (result.args._message)
    )
  )))
```

Note how we invoked `Record({}, {fromBlock: 0, toBlock: "latest"})`. The first argument is used for filtering the results, which is outside the scope of this post. The second one is how we specify which block span we'll be looking for the transaction logs for `Record`.

The `get` method takes a callback, and we're just `map` ping over the the returned result to get to the `args._message` value of each result.

If this all worked out, you should see your console output the following.

```
ℹ️ truffle console
------------------

truffle(development)> [ 'first message', 'second message']
```

And that's it! Your contract is deployed, you've used it to write messages into the chain, and you then queried it to get those messages back.

Testing

Before we try this on real networks, let's look into testing the contract first. Truffle uses the Mocha testing framework, and Chai for assertions. We'll be using the following test.

```
ℹ️ test/recorder.js
-------------------

var Recorder = artifacts.require("Recorder");

contract('Recorder', function(accounts) {
  it("records event", function(done) {
    Recorder.deployed().then(function (instance) {
      instance
        .Record({}, {fromBlock: 0, toBlock: "latest"})
        .get(function (error, events) {
          assert.equal(
            0,
            events.length,
            "There should be no events at start."
          );
          instance.record("pokemon").then(function () {
            instance
              .Record({}, {fromBlock: 0, toBlock:
"latest"})
              .get(function (error, events) {
                assert.equal(
```

```
                    1,
                    events.length,
                    "There should be one event after
recording."
                  );
                  done();
                });
            });
          });
      });
    });
});
```

It looks ugly, but it’s pretty simple as far as tests go:

- `Recorder.deployed()` resolves when it gets a hold of the deployed Recorder instance
- we use `instance.Record(...).get()` first, to get the list of messages written to it
- we check that it has no messages, with `assert.equal(0, events.length)`
- we then write one message with `instance.record("pokemon")`
- and finally, we ensure the message was written, with `assert.equal(1, events.length)`

Create a file in the `tests` directory called `recorder.js` and paste the test into it. You can run this test by exiting the console and simply typing `truffle test` into your shell. Note that truffle deploys the contract anew every time you run a test file. See the documentation for more details.

```
ℹ️ truffle test output
----------------------
```

```
Using network 'development'.

Contract: Recorder
    ✓ records event (101ms)
```

```
  1 passing (115ms)
```

Success!

Connecting to a real network

You can now legitimately say that *it works on my machine*. Not good enough? Let's try and get this to work on an actual Ethereum network.

Before we deploy into the actual *mainnet*, let's use a *testnet* first. A testnet is an Ethereum network analog, supported by real nodes, but used mostly for testing. As such, testnets aren't used for anything serious and the Ether therein shouldn't hold any value. We'll be using the Ropsten testnet. This will allow us to use a block explorer to dig for our contract and transactions on the network.

Install geth as per the instructions. Then we'll sync up with Ropsten (more instructions).

```
ℹ️ shell
--------

$ geth --testnet --fast --bootnodes
"enode://20c9ad97c081d63397d7b685a412227a40e23c8bdc6688c6f3
7e97cfbc22d2b4d1db1510d8f61e6a8866ad7f0e17c02b14182d37ea7c3
c8b9c2683aeb6b733a1@52.169.14.227:30303,enode://6ce05930c72
abc632c58e2e4324f7c7ea478cec0ed4fa2528982cf34483094e9cbc921
6e7aa349691242576d552a2a56aaeae426c5303ded677ce455ba1acd9d@
13.84.180.240:30303" --rpc
```

- `--testnet` tells geth to connect to the Ropsten testnet
- `--fast` or `--syncmode="fast"` makes the blockchain synchronization faster (see this PR for more details);

- `--bootnodes=...` tells geth which nodes to connect to in order to discover the rest of the network
- `--rpc` tells geth to enable the HTTP-RPC server, allowing trufle and other RPC clients to connect to it

We can use the console geth provides to interact with the network. `geth attach` should do the trick, but if it doesn't, look for the following line in the geth output.

```
ℹ️ geth output
--------------

INFO [11-13|14:45:16] IPC endpoint opened:
/Users/pokemon/.rinkeby/geth.ipc
```

And run `geth attach /Users/pokemon/.rinkeby/geth.ipc` accordingly. This should drop you right into the console. Geth should be hard at work syncing with the Ropsten chain, which should take anywhere from a few minutes to a few hours, depending on your hardware and network. Here's how we can check progress.

```
ℹ️ geth console
---------------

> eth.blockNumber
0
```

`eth.blockNumber` returns the number of the most recent block [docs]. It'll display `0` until you've synced up with the network, which we can check with `eth.syncing` :

```
ℹ️ geth console
---------------

> eth.syncing
false
```

Bummer. It seems we aren't connected to any nodes yet. We'll have to wait to see this in geth's output before we can continue.

```
ℹ️ geth output
--------------

INFO [11-13|15:26:26] Block synchronisation started
INFO [11-13|15:26:28] Imported new block headers
count=384 elapsed=951.820ms number=384 hash=d3d5d5…c79cf3
ignored=0
```

Great! Let's try the console again:

```
ℹ️ geth console
---------------

> eth.blockNumber
0
> eth.syncing
{
  currentBlock: 775342,
  highestBlock: 2356143,
  knownStates: 167957,
  pulledStates: 162021,
  startingBlock: 0
}
```

Much better. We'll wait until `eth.syncing` returns `false` again before we interact with the blockchain. That 0 up there is ok, don't worry.

Here's a few other commands to keep you busy:

- `net.listening` tells us whether the node is actively listening for network connections or not [docs]
- `net.peerCount` returns the number of connected peers [docs]
- `admin.peers` gives us insight into the peers we're connected to [docs]

. . .

Let's try deploying our contract to the Ropsten network. We'll let Truffle do the hard work for us, again. However, Ropsten being a genuine faux network, we'll need to get some genuine faux Ether for one of our accounts.

Let's head on over to geth and confirm we have 0 Ether first:

```
ℹ️ geth console
---------------

> eth.getBalance(eth.accounts[0])
0
```

Yep. I tried using a Ropsten Faucet but it was down when I tried it. The Ropsten Gitter recommended I use MetaMask's Ether Faucet, which you need MetaMask to use, and it worked well for me.

I then used MetaMask to send Ether to my account in geth. First I had to find out its address.

```
ℹ️ geth console
---------------
```

```
> eth.accounts[0]
"0xff7771583ee3944a9ef32cfcb54c0b0d688fa70a"
```

Then I used “Send” inside MetaMask to send funds to this address. The transfer shouldn’t take more than a couple of minutes (you can click the transfer in MetaMask to go check it out on Etherscan). Once it’s confirmed, you should be able to confirm this inside geth:

```
ℹ️ geth console
---------------

> eth.getBalance(eth.accounts[0])
1836311700000000000
```

Let’s also use this account address to tell Truffle which account to use to interact with the blockchain. Update your `truffle.js` file to look like this:

```
ℹ️ truffle.js
-------------

module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "*",
      from: /* YOUR ADDRESS HERE */,
      gas: 4000000,
    }
  }
};
```

Because we’re sticklers, we’re running the tests again, now on Ropsten.

```
ℹ️ shell
--------

$ truffle test

Using network 'development'.

Error encountered, bailing. Network state unknown. Review
successful transactions manually.
Error: authentication needed: password or unlock
```

Oh, right. So, geth keeps this account locked for your protection. In order to use it, you need to unlock it. Back to geth: let's keep the account unlocked as long as geth remains open [docs].

```
ℹ️ geth console
---------------

> personal.unlockAccount(eth.accounts[0], "password", 0)
```

Great. Let's try again now.

```
ℹ️ shell
--------

$ truffle test

Using network 'development'.

Contract: Recorder
    ✓ records event (54382ms)

1 passing (54s)
```

It took a lot longer than with testrpc, but the tests passed. Now let's boot `truffle console` and try to deploy the contract. This time, testrpc is not the target. Ropsten is.

```
ℹ️ truffle console
------------------

truffle(development)> migrate --reset
Using network 'development'.

Running migration: 1_initial_migration.js
  Replacing Migrations...
  ...
0x2b330d5a4bac42ac3aea135c0f7090502f05e5349a6684c869db9a959
1d02bd3
```

Much better, but this will take a while to finish. Take the time to note something like this on geth's output.

```
ℹ️ geth output
--------------

INFO [11-22|11:05:50] Submitted contract creation
fullhash=0xfb92a0da5e5b7d13a155b1c57bb46ffeb672502842ffe744
de30be068a4d5b4f
contract=0xf7Ec3C971DFA6549e2c0aC4437068e86dC39b25F
```

Neat, right? By the way, you can copy that transaction hash and contract address and go look for them in Ropsten Etherscan. It might take a minute for it to show up, but it will. We're deploying contracts onto a real network now. Remember, you don't own a server. This is nothing short of awesome.

In the meantime, Truffle keeps going.

```
ℹ️ truffle console
------------------

Running migration: 1_initial_migration.js
  Replacing Migrations...
  ...
0x2b330d5a4bac42ac3aea135c0f7090502f05e5349a6684c869db9a959
1d02bd3
  Migrations: 0xf1867e8ac0fe8c4dd26157c2e5123b9f5d63e0e0
Saving successful migration to network...
  ...
0x3c0364b133a07bed6f34c1e2b733f57cfee65ded088bdd452547bfba5
eb3c18c
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Recorder...
  ...
0x4c86ec2e5da57f734706257a86e7fff929538a68ed7a5265cc628b984
613d822
  Recorder: 0x949d9ac08845f74ec7cd96337920b672fb31c017
Saving successful migration to network...
  ...
0x02c04fc250d3077c12eef7e3c580564f03791ee53de2a1dfcc82eb37d
eba3911
Saving artifacts...
```

Sweet. We're done. See this line?

```
ℹ️ truffle console
------------------

Recorder: 0x949d9ac08845f74ec7cd96337920b672fb31c017
```

Yea, that's us. We're live on the Ropsten testnet.

Let's play with our contract. We know it works because it passes the tests, but still. Back to Truffle, so we can record a message and make sure it stays there.

```
ℹ️ truffle console
------------------

truffle(development)> recorder =
Recorder.at(Recorder.address)
truffle(development)> recorder.record("i dont even")
```

This will take a couple of minutes, since we're on Ropsten now. But then:

```
ℹ️ truffle console
------------------

truffle(development)> recorder.Record(
  {},
  {fromBlock: 0, toBlock: "latest"}
).get(function (error,result) {
  console.log(result.map((x)=>(x.args._message)))
})
truffle(development)> ['i dont even']
```

Perfect!

Using the Ethereum mainnet

Armed with all this knowledge, we can now easily deploy the same contract into the mainnet. For brevity, we'll skip a few steps like testing, and get right to it.

Let's restart geth and have it connect to the mainnet.

```
ℹ️ shell
--------

$ geth --fast --cache=512
```

We'll need to wait for geth to sync up. Go get busy elsewhere, this will take some time now.

Once `eth.syncing` returns false again, let's use truffle to get this deployed. Make sure that whichever account you're using has funds in it. Real ether.

```
ℹ️ shell
--------

$ truffle migrate --reset
```

This will probably not go well at first. There's two failure modes that you're likely to run into. There's this beauty:

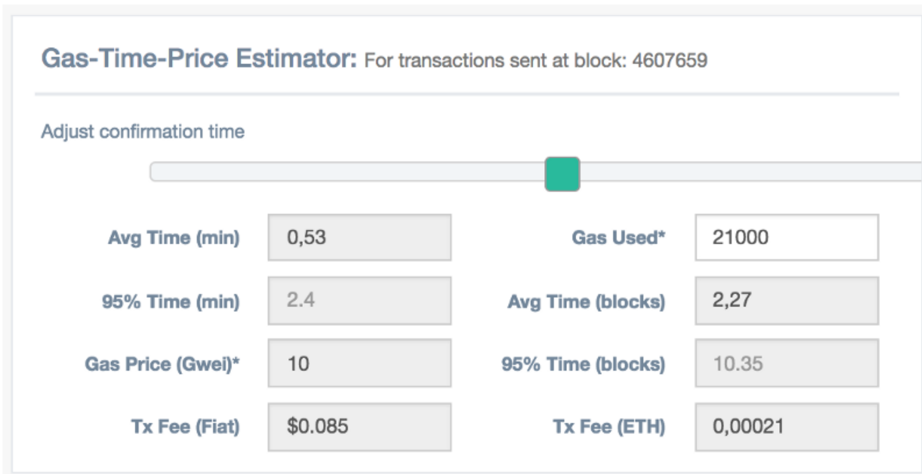
```
ℹ️ truffle migrate output
-------------------------

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... undefined
Error encountered, bailing. Network state unknown. Review
successful transactions manually.
Error: insufficient funds for gas * price + value
```

And this one too:

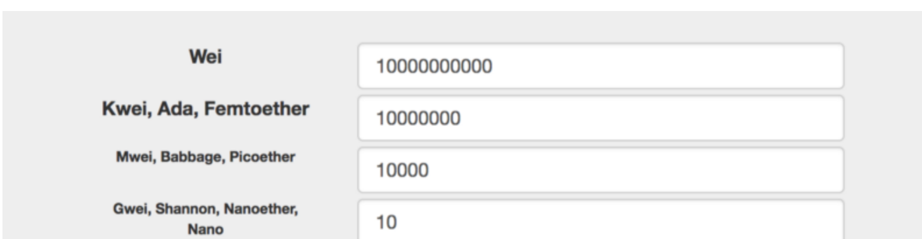
```
ℹ️ truffle migrate output
-------------------------

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... undefined
Error encountered, bailing. Network state unknown. Review
```

```
successful transactions manually.
Error: exceeds block gas limit
```

So, this is geth telling truffle that your gas settings aren't ideal. I recommend setting `gas` to a reasonably high value, like 1000000 (notice above how the contract deployment was using up about ~20K gas), and checking out ETH Gas Station to figure out how much the value of `gasPrice` should be. Here's what I see now:



So, 10 Gwei gets us a confirmation in 0.53 minutes. That's pretty good. Ethereum Converter can helps us turn that into wei.



So here's what truffle.config should look like now.

```
ℹ️ truffle.js
-------------
```

```
module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "*",
      from: "0x13f53d42fc7cf4f1cf4ca8031a526f6a8528cdfa",
      gas: 1000000,
      gasPrice: 10000000000,
    }
  }
};
```

This should have you well on your way!

Building a web frontend

Let's move on to implementing a web frontend for this contract. Sure—it's already public and available for anyone to use… kind of. Using `geth` isn't the most user-friendly tool, plus you can't easily interact with a contract you don't have the ABI for. So, we'll need to go the extra mile to make this available to the general public.

We'll start by using web3.js and connecting it to our local node. What's web3.js? From the README:

> *This is the Ethereum compatible JavaScript API which implements the Generic JSON RPC spec.*

Let's go back to testrpc, since it's faster than using a real network. Start it with `testrpc` . Re-adjust your `truffle.js` , and re-deploy your contract. Get its address.

Then get the first account address from the `testrpc` output.

```
ℹ️ testrpc output
-----------------
```

```
Available Accounts
==================
(0) 0xa2dee6cd83d3d9cb6fb2d42899abcbdf04bf344f
```

We can use Truffle to check the balance of this account. To do that, we'll also use the web3 library, which is conveniently included in truffle. It tells us this account is loaded.

```
ℹ️ truffle console
------------------

truffle(development)> account =
"0xa2dee6cd83d3d9cb6fb2d42899abcbdf04bf344f"
truffle(development)>
parseInt(web3.eth.getBalance(account))
'99944815499999920000'
```

Great! Let's double-check that in the browser. Download web3.js and include it on an HTML file with a `<script>` tag. Open that file in your browser and throw this in the console.

```
ℹ️ browser javascript console
-----------------------------

> var host = "http://localhost:8545"
> var web3 = new Web3(new
Web3.providers.HttpProvider(host))
> var account =
"0xa2dee6cd83d3d9cb6fb2d42899abcbdf04bf344f"
> parseInt(web3.eth.getBalance(account))
< 99944815499999920000
```

Great. Now that we know we can connect our browser to testrpc, let's move on to better things. What we want to do is test writing to and reading from the contract. Here's a start.

```
ℹ️ browser javascript console
-----------------------------

> var contractABI =
JSON.parse('[{"constant":false,"inputs":
[{"name":"message","type":"string"}],"name":"record","outpu
ts":
[],"payable":false,"stateMutability":"nonpayable","type":"f
unction"},{"anonymous":false,"inputs":
[{"indexed":false,"name":"_from","type":"address"},
{"indexed":false,"name":"_message","type":"string"}],"name"
:"Record","type":"event"}]');
> var contractAddress =
"0x2680998ab6aa13fd092636ec974f5e305a8d9051";
> var web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
> web3.eth.defaultAccount =
"0xa2dee6cd83d3d9cb6fb2d42899abcbdf04bf344f";
> var contract =
web3.eth.contract(contractABI).at(contractAddress);
```

Alright, we have a hold of a `contract` instance now. Let's try and record something onto it.

```
ℹ️ browser javascript console
-----------------------------

> contract.record("hello from console");
<
"0x09506da426e48b5b9a36cda4dd176767a1a3a1392c80299a4b3a1628
1f10ece3"
```

Oh look, it's a transaction ID. This has got to be good. Let's check that it worked, shall we?

```
ℹ️ browser javascript console
-----------------------------

> contract.Record({}, {fromBlock: 0, toBlock:
"latest"}).get(function (error, results) {
```

```
  console.log(results.map((result) =>
(result.args._message)));
});
["hello from console"]
```

It seems to work! We're now ready to move on.

Using MetaMask

Notice we're setting `web3.eth.defaultAccount` above, and using one of the accounts testrpc created for us. Now, this is fine for development, but when you throw this on the Web, you'd ideally like users to be the ones paying for the transactions.

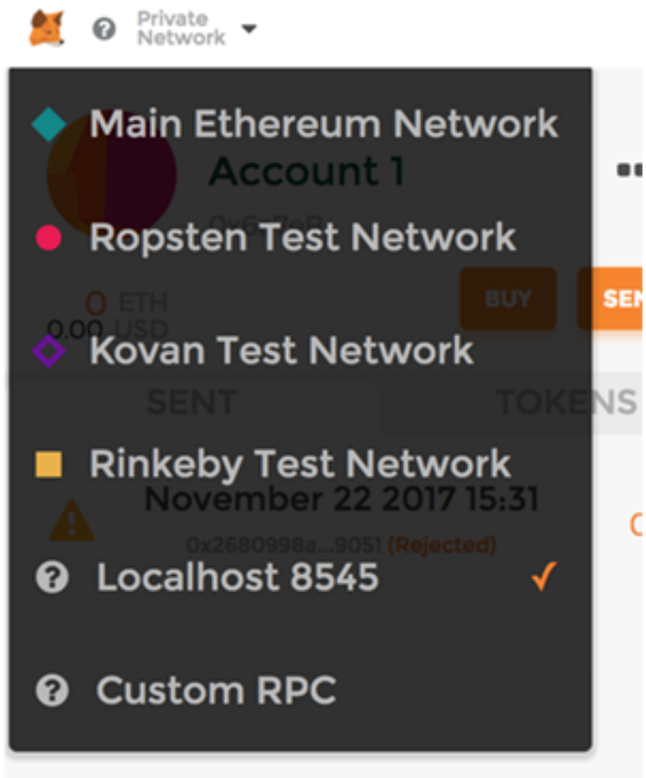
Enter MetaMask. MetaMask is a Chrome extension serving as an in-browser Ethereum wallet. **You'll need it to continue, so go install it.**

Additionally, to use MetaMask for development, you'll need to have a local server running, as just opening the HTML file won't cut it. Do that in any way you'd like. I like using python.

```
ℹ️ shell
--------

$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Get rid of the `<script>` tag and head on over to localhost:8000 on your browser. If you type `web3` into the console, you shouldn't see `undefined`. If you do, make sure MetaMask is running.

Finally, ensure MetaMask is connected to your local node.



Let's try this again, now without setting `web3.eth.defaultAccount`. Throw this in the console.

```
ℹ️ browser javascript console
-----------------------------
> var contractABI =
JSON.parse('[{"constant":false,"inputs":
[{"name":"message","type":"string"}],"name":"record","outpu
ts":
[],"payable":false,"stateMutability":"nonpayable","type":"f
unction"},{"anonymous":false,"inputs":
[{"indexed":false,"name":"_from","type":"address"},
{"indexed":false,"name":"_message","type":"string"}],"name"
:"Record","type":"event"}]');
> var contractAddress =
"0x2680998ab6aa13fd092636ec974f5e305a8d9051";
var web3 = new Web3(web3.currentProvider);
var contract =
web3.eth.contract(contractABI).at(contractAddress);

contract.Record({}, {fromBlock: 0, toBlock:
"latest"}).get(function (error, results) {
  console.log(results.map((result) =>
```

```
(result.args._message)));
});
["hello from console"]
```

It works as expected. Let's try and record another message and make sure we can read that one as well.

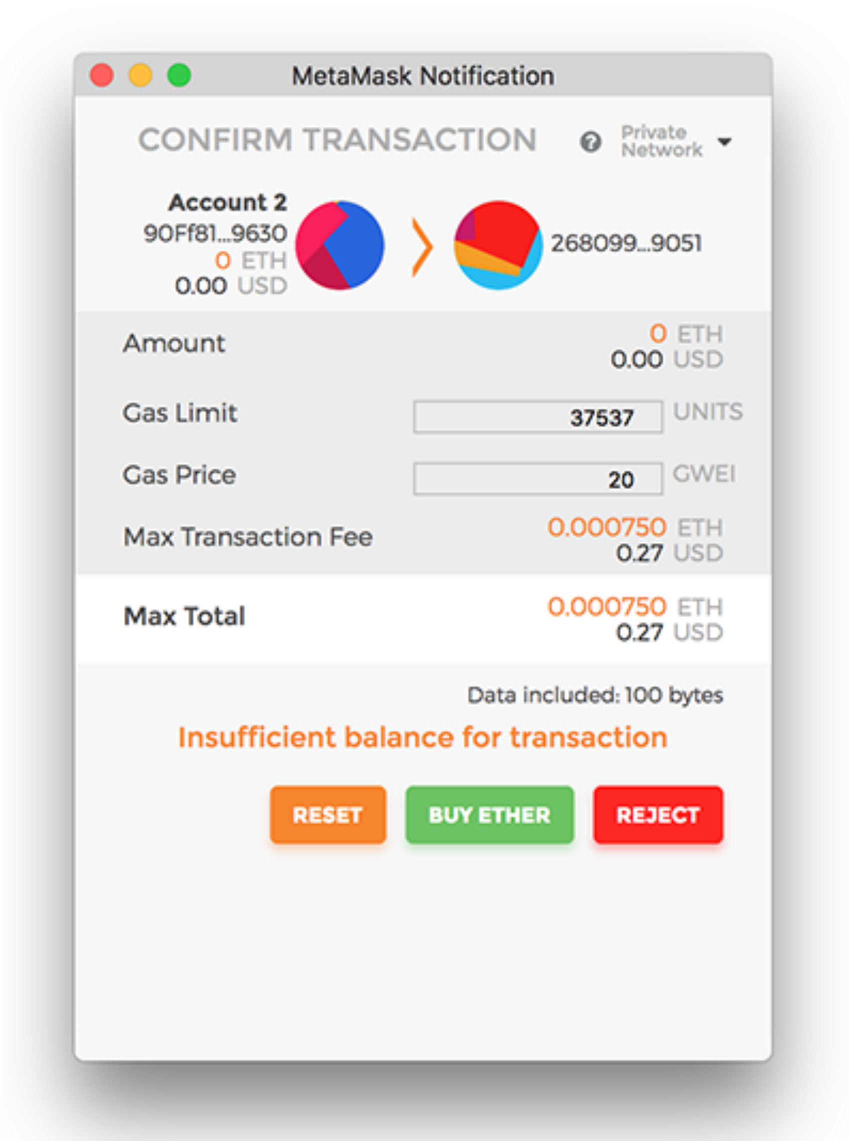
```
ℹ️ browser javascript console
-----------------------------

> contract.record("hello again")
Uncaught Error: The MetaMask Web3 object does not support
synchronous methods like eth_sendTransaction without a
callback parameter
```

Uh-oh. This happens because MetaMask is a light Ethereum client. This means that it doesn't store all of the blockchain data, and depends on asking the network for the data it needs every time.

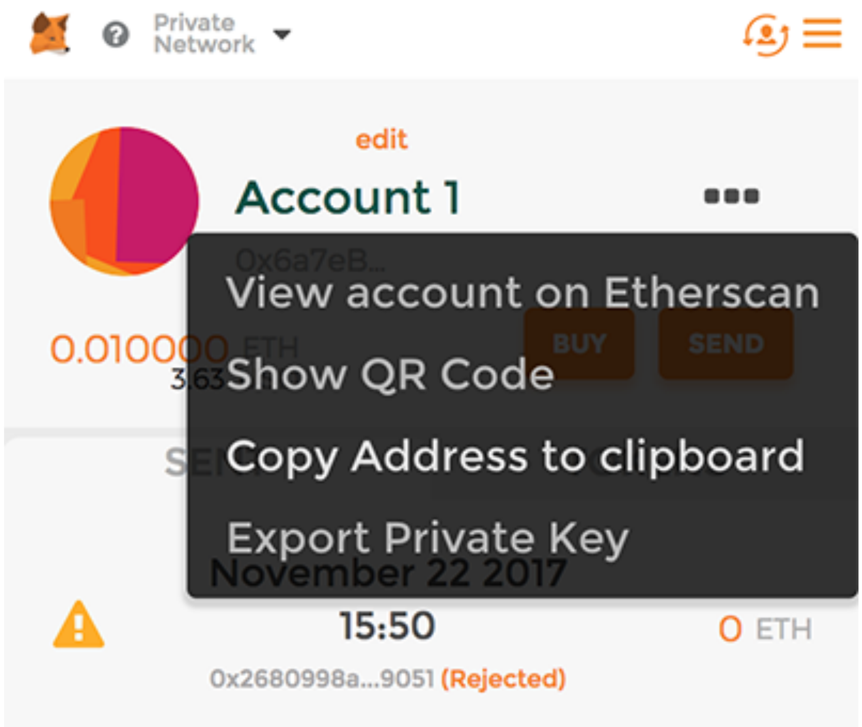
As such, we'll need to use asynchronous methods to do the same thing we were doing before. Not too complicated, mind you.

```
ℹ️ browser javascript console
-----------------------------

> contract.record("hello again", (error, result) =>
(console.log(result)))
```

Once you've done this, MetaMask should pop this up.



Note that it says your balance is insufficient. That makes sense. Let's send some funds to this account that MetaMask created for us. First, get a hold of the account address.



Now, back to truffle.

```
ℹ️ truffle console
------------------

truffle(development)> web3.eth.sendTransaction({
  from: web3.eth.accounts[0],
  to: "0x6a7eB27407a50a4eb9d015EA2B0F2e1BcC724461",
  value: 5000000000000000}
)
```

Let's try the same command again now.

```
ℹ️ browser javascript console
-----------------------------

> contract.record("hello again", (error, result) =>
(console.log(result)))
```

This time, the popup should have no error messages. Click send. You should be able to see something like this on testrpc.

```
ℹ️ testrpc output
-----------------


eth_sendRawTransaction

Transaction:
0xfd78adcb6bd3aa86d6e67e5edc946ca3b53235553d4cc5a409c3566bd
afb13f6
  Gas usage: 25025
  Block Number: 24
  Block Time: Wed Nov 22 2017 15:58:27 GMT+0100 (CET)
```

Did it work? Well, let's check it out!

```
ℹ️ browser javascript console
-----------------------------


> contract.Record({}, {fromBlock: 0, toBlock:
"latest"}).get(function (error, results) {
  console.log(results.map((result) =>
(result.args._message)));
});
["hello from console", "hello again"]
```

That's 2 messages there. Success!

From here on out, you'd switch from testrpc to Ropsten and then the Ethereum mainnet. I'll spare you this duplication in this post.

Final notes

There's a couple of extra things we had to do (aside from, well, building the website) in order to get Forever on the Chain working as intended.

In order to have the "Saved messages" list update itself with new entries, we decided to make use of the event watch method. This sets up a listener which invokes your callback whenever an event, fitting of whichever criteria you like, lands on the chain. Here's how we did it.

```
contract.Record(
  {},
  {fromBlock: 4491369, toBlock: "latest"}
).watch(function (error, result) {
  // DO THINGS
});
```

Note that `4491369` up there. Since we're connecting to mainnet, sweeping all transactions from block 0 would be prohibitively slow. As such, we're limiting our search to events starting at the block number of the block that the contract was deployed into. You can find that number by using Etherscan, but web3 can also help you if you know the transaction the contract was created in.

Even though this is pretty straightforward, MetaMask wasn't happy with this, so we ended up deploying a light node on an EC2 instance instead.

The code for this project is available on GitHub.