Ethereum programming for web developers

by Jon Evans, CTO, HappyFunCorp
jon@happyfuncorp.com

Hello, fellow web developer! If you're reading this, you're probably interested in blockchains, smart contracts, etc., as someone who actually wants to write some smart-contract code. I'm going to walk you through setting up, writing, and deploying a smart contract to a real live Ethereum blockchain, and then interacting with that contract in a browser via a web service.

The Ethereum Virtual Machine

I'm not going to explain Blockchains 101 or Ethereum 101: there are many other places to go for that. But it's probably worth discussing Ethereum at a very high level from a developer's perspective.

You don't need to care about mining or Proof-of-Work vs. Proof-of-Stake, or anything like that. But you should know that Ethereum is a decentralized virtual machine that runs on many nodes scattered around the world, and so-called "smart contracts" are code which runs (along with data which is stored) within that virtual machine, i.e. on every single node.

This is obviously hugely inefficient, but it has advantages; everyone in the world can rely on this code/data, because no central service or system can tamper with it; and anyone can submit code/data to this machine without the registering or asking permission. They do, however, need to pay. Every line of code and byte of storage in Ethereum has a price.

Ethereum, like Bitcoin, has a native currency, called "ether"; this is the same Ether currency that is traded on exchanges like Coinbase. When used to pay for Ethereum computing/storage, it is called "gas." For any given smart contract, gas has a "limit" and a "price." This is pretty confusing at first, but don't worry, you'll wrap your head around it eventually, and anyway this tutorial uses free fake money on a so-called "testnet" Ethereum blockchain.

Blockchain Languages

In principle many languages can be compiled down to the bytecode used by the Ethereum VM, but in practice almost all smart contracts are written in the "Ethereum-native" language called Solidity. Solidity is still arguably somewhere between alpha-release and beta-release quality, and has lots of ... idiosyncracies. (See this scathing commentary from six months ago: https://news.ycombinator.com/item?id=14691212) Still, it remains the de facto state of the art.

Solidity reads a bit like Javascript. It has a lot of quirks and pitfalls, though, especially when it comes to moving money around. Be *very* careful if writing real money-transfer code; do your security homework, get others to review your code, and seriously consider an official security audit or even formal verification. Again, this tutorial uses fake/test money, not real money.

Solidity code runs *on* the blockchain. But talking *to* the Ethereum blockchain from external code -- like, say, a web server -- is another matter. In theory, you could roll your own JSON-RPC calls; in practice, you want to use an existing library. The de facto state of the art here is a Javascript library called "web3". I'm more of a Ruby / Python / Go server-sider myself, but (for better or worse) JS is widely used and understood by web developers everywhere, so much of this tutorial is written in Node. Reminder: "the web framework named Node" and "an Ethereum node" are two completely different uses of the same word. You have been warned.

Most Solidity tutorials assume you're running an Ethereum node on your machine, and/or one or more browser plugins. These steps add complexity & cognitive overhead and are not actually necessary. This tutorial is for web developers, who are accustomed to talking to APIs running on external servers; it will get your smart contracts up and running, on a real live blockchain, without ever needing to run an Ethereum node yourself.

Overview

Let's quickly go over what we're about to cover:

1. **Selecting an Ethereum testnet; creating an address; storing its private key; financing it with fake money**
2. **Installing and understanding the Truffle development environment for Solidity**
3. **Writing your first smart contract**
4. **Testing your first smart contract**
5. **Deploying your first smart contract**
6. **Talking to your smart contract from a Node web service**
7. **End-to-end browser <> Node <> blockchain <> Node <> database data flow.**

Also, I'm a believer in entire tutorials contained within a single document, which this is. That said, you can also go look at the Git repository containing the final code at https://github.com/HappyFunCorp/Webthereum and witness it up and running (crudely) on Heroku at https://webthereum.herokuapp.com/ -- though please *please* read the security discussion at the very end of this document first, or, at least, don't enter real private keys into that app. (It would *probably* be safe to do so -- enforced HTTPS connection, they're never saved on the server side, etc. -- but that's not the point.)

1. Testnet setup

The "real" Ethereum blockchain is protected by a vast network of miners securing its transactions with billions of hashes per second. However there are also "testnet" blockchains, which are either less secure or are "private" i.e. controlled by a small subset of permissioned miners. We will use the Rinkeby (https://www.rinkeby.io/) testnet, an Ethereum blockchain that anyone can connect to remotely, courtesy of the fine folks at Infura (http://infura.io/).

But first you need an address and private key. You can go make these online too. Just head over to MyEtherWallet (https://www.myetherwallet.com/) and enter a password. You won't use this for real money, so don't worry too much about security. If you *were* using this for real money, you'd be much, much more careful about this process, though, right?

Once you've entered your password, you will be offered the opportunity to "Download Keystore File." Do so.

Then go back to MyEtherWallet, click on "View Wallet Info", select "Keystore / JSON file," re-upload the file you just downloaded, and re-enter the password you just entered. Two important strings of alphanumeric characters will be revealed to you: your *address* and your *private key*. (Click on the little eye icon to reveal the latter.)

Again, if you were dealing with real amounts of real money, and/or important smart contracts, you would take significant measures to protect and secure your private key. You aren't -- but, still, you don't want to get into the habit of hardcoding your private key into your code and/or your repo.

I strongly suggest that you make the address and the private key *environment variables*. Then later you can deploy your code to AWS or Heroku or GCE and simply set the appropriate environment variables there. This also makes it easy to have different testing / staging / production environments.

In a Bash environment like OS X, you can do this by appending these two lines to your .bash_profile in your home directory:

```
export RINKEBY_ADDRESS="0x1234567890ABCDEF..."
export RINKEBY_KEY="ABCDEF123456789..."
```

Setting environment variables in other environments is left as an exercise for the reader. Note that you'll have to close and re-open your terminal window to refresh your environment. (Check with "printenv" in bash.)

Voila: you now have a valid Ethereum address, and the private key used to sign its transactions, available for use in your code. However, to actually deploy and run transactions, you'll need (fake) money. Conveniently you can get this online, too, via the Rinkeby faucet:

https://faucet.rinkeby.io/

All you need to do is post your new Ethereum address (*not* your private

key!) to Facebook or Twitter or Google+ (publicly): paste a link to that tweet/post into the input box at that faucet's web page; and wait at most a minute or two. The minimum amount of ether you'll receive (3) is already more than you need for basic development, but still, might as well get as much as you can.

The faucet transaction might take a minute to complete. You can check your balance via Infura's Etherscan service:

https://rinkeby.etherscan.io/address/[your-address-goes-here]

Success? Congratulations! It is now *almost* time to move on to writing some code. First, though, we have one more thing to install and configure. Don't worry, it's easy; trust me, it's worth it.

2. Truffle

First -- you're familiar with Javascript, right? If not ... this may not be the tutorial for you. OK, you're vaguely familiar with Node and NPM, right? If not, get thee first to a different tutorial (eg http://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/) and study up.

Know your away around those basics? Great. On to the Truffle Framework (http://truffleframework.com/docs/), "your Ethereum Swiss Army Knife." You'll use it to compile and test your Solidity code. You can also use Truffle to deploy your code, but it needs a local Ethereum node, so we'll deploy in a different way, described below.

Installing Truffle via NPM is as easy as

```
npm install -g truffle
```

It then offers various commands, the most relevant of which for this tutorial are "truffle init", "truffle compile", and "truffle test." Once installed, open up a terminal window, create a new directory for your new Ethereum project -- I suggest the name "webthereum" -- go to it, create a "truffle" subdirectory, go to *it*, and type

truffle init

Voila: a scaffolding of files and directories will be created for you.

Now, open up the text editor of your choice and create a new file called "MyPrize.sol" in the "contracts" sub-subdirectory that Truffle just created for you. It's finally time to start writing yourself some smart contracts.

3. Baby's First Smart Contract

Our smart contract, MyPrize, is going to manage ownership of 10 geolocated augmented-reality entities, called "prizes." Each prize has an owner; a location; and a "metadata" field for a URL which points to the metadata that describes its contents, i.e. what it looks like in AR. For our smart-contract purposes, all we care about is that there's a string called "metadata."

After a prize has been placed, it stays where it is until 1000 new blocks have been added to the blockchain (i.e. about 4 hours), and then anyone else can claim it, rewrite the metadata (e.g. what this entity looks like you point a custom AR app at the location where it has been placed) and then place it somewhere else. It's up to some external system (e.g. a custom Android/iOS/web app talking to a web service) to handle parsing the metadata, showing the AR object, etc. Our smart contract just handles ownership and location. Simple enough. Right?

There can be many subtle complexities when writing Solidity smart contracts. Contracts can send and receive money. Contracts can spawn, own, and link to other contracts. Issues of identity, ownership, and versioning come up. It's common for whoever first deploys a contract to be its "owner," and for important functionality to be calved off to separate subcontracts, which the owner can replace if a bug needs to be fixed or when a new version is ready.

Does this "code written so that its owner, and/or a set of other designated admins, can control, replace, and disable it" pattern play a little awkwardly with the "smart contracts are permissionless and irrevocable with no central control" hype? You betcha! But we're not

going to worry about theoretical philosophical concerns; we're just going to write some running code. And here it is:

```
pragma solidity ^0.4.18;

contract MyPrize {
  struct Prize {
    uint id;
    address owner;        // current owner
    bytes10 geohash;      // current unclaimed location
    string metadata;      // probably a URL pointing to the full metadata
    uint placedAt;        // block count when it was placed
  }

  mapping (uint => Prize) public prizes;

  // Constructor
  function MyPrize() public {
    for (uint i=1; i <= 10; i++) {
        Prize memory prize = Prize({id: i, owner: msg.sender, geohash: "",
metadata: "", placedAt: 0});
        prizes[i] = prize;
    }
  }

  function placePrize (uint prizeId, bytes10 geohash, string _metadata) public
{
    var prize = prizes[prizeId];
    require (prize.id != 0);
    require (geohash != bytes10(""));
    require (prize.owner == address(0) || msg.sender == prize.owner);
    prize.owner = address(0);
    prize.geohash = geohash;
    prize.metadata = _metadata;
    prize.placedAt = block.number;
  }

  function claimPrize (uint prizeId, bytes10 geohash) public {
    var prize = prizes[prizeId];
    require (prize.id != 0);
    require (geohash != bytes10("") && prize.geohash == geohash);
    require (block.number - prize.placedAt > 1000);
    prize.owner = msg.sender;
    prize.geohash = "";
    prize.metadata = "";
```

```
    prize.placedAt = 0;
  }
}
```

Save that file, go back to your command line, go back to the "truffle" subdirectory beneath your root "webthereum" directory, and type

truffle compile

You should see something like:

Compiling ./contracts/MyPrize.sol...
Writing artifacts to ./build/contracts

4. Testing Baby's First Smart Contract

Writing automated tests for your software is always a good idea; but it's a *really, really excellently awesome and important idea* for Solidity code. If you don't find a bug until the code is on the blockchain, not only have you wasted a ton of time and a slew of money, your diagnosis and debugging options are very, very limited. Did I mention that they are expensive and time-consuming, too?

So test. To its credit, Truffle lets you write test in both Solidity and Javascript. You could make a good case for writing both kinds of tests (so that you test both the VM operations, with the former, and the JS-blockchain interface) but in the interests of expediency let's just whip up a Solidity test for MyPrize, called TestMyPrize.sol, within the "test" sub-subdirectory Truffle created for us:

```
pragma solidity ^0.4.18;

import "truffle/Assert.sol";
import "../contracts/MyPrize.sol";

contract TestMyPrize {
  MyPrize mp = new MyPrize();

  function testMyPrize() public {
    mp.placePrize(1, bytes10("abcdefghij"), "http://test.example.com/");
    var (id, owner, geohash, metadata, placedAt) = mp.prizes(uint(1));
```

```
    Assert.equal (bytes10("abcdefghij"), geohash, "Geohash was successfully
set");
    Assert.equal (address(0), owner, "Ownership was successfully abdicated");
    Assert.equal (block.number, placedAt, "PlacedAt successfully set");

    // mp.claimPrize(1, bytes10("abcdefghij"));
    // var (id2, owner2, geohash2, metadata2, placedAt2) =
mp.prizes(uint(1));
    // Assert.equal (bytes10(""), geohash2, "Geohash was successfully
cleared");
    // Assert.equal (msg.sender, owner2, "Ownership was successfully
claimed");
  }
}
```

You'll notice that we don't test the metadata, and that the "claim" tests are commented out. This is because we can't actually test either of those things; Truffle's Assert.equal doesn't work with strings returned from contract calls, and because (as far as I can tell) we can't easily advance the block number within Truffle. So the claim call will fail ungracefully if we try, teaching us nothing. I did mention that the whole ecosystem is still somewhere between alpha and beta software, right?

But we test what we can. Back to the command line, and

truffle test

Et voila!

```
  TestMyPrize
    ✓ testMyPrize (105ms)

  1 passing (691ms)
```

5. Deploying Baby's First Smart Contract

So you wrote (well, you copy-and-pasted, but it's a start) your first smart contract. You tested your first smart contract. You've got a blockchain address and a private key. Now how do all these ingredients bake together into a delicious cake running in production? (Pardon the mixed metaphor; I'm hungry as I type this.)

You could run an Ethereum client like Geth or Parity or MetaMask on your local machine, point Truffle to it, and simply type "truffle deploy". But let's go a little more low-level, and at the same time, keep you from having to run your own local node. Infura lets you submit signed Ethereum transactions to the Rinkeby testnet via the Internet, and contract deployment is just another kind of Ethereum transaction.

But you don't want to have to deal with a slew of cryptic bash commands either. Good news: you don't need to. You can compile and deploy entirely via Javascript, courtesy of the web3 library and the solc compiler. Let's take a look at that in a little more detail:

First off, create a new "node" subdirectory under your root "webthereum" directory. Then go there, run

npm init

and tap ENTER through all the default values, or tweak them as you like. (You do have NPM installed and running on your system, right?) Next, install a few new NPM packages:

npm install web3
npm install solc
npm install fs
npm install ethereumjs-tx

And now, save the following code to "deploy.js" in that newly created "node" directory:

```
var Web3 = require('web3');
var solc = require('solc');
var fs = require('fs');
var tx = require('ethereumjs-tx');

const web3 = new Web3(new
Web3.providers.HttpProvider("https://rinkeby.infura.io/"));

deployContract();
// showABI();

function deployContract() {
```

```
web3.eth.getTransactionCount(process.env.RINKEBY_ADDRESS).then((txnCou
nt) => {
    console.log("txn count", txnCount);
    const source =
fs.readFileSync(__dirname+'/../truffle/contracts/MyPrize.sol');
    const compiled = solc.compile(source.toString(), 1);
    const bytecode = compiled.contracts[':MyPrize'].bytecode;
    const rawContractTx = {
        from: process.env.RINKEBY_ADDRESS,
        nonce: web3.utils.toHex(txnCount),
        gasLimit: web3.utils.toHex(4000000),
        gasPrice: web3.utils.toHex(20000000000),
        data: '0x' + bytecode,
    };
    console.log("contract deploying...");
    sendRaw(rawContractTx);
  });
}

function showABI() {
    const source =
fs.readFileSync(__dirname+'/../truffle/contracts/MyPrize.sol');
    const compiled = solc.compile(source.toString(), 1);
    const abi = compiled.contracts[':MyPrize'].interface;
    console.log("abi", abi);
}

function sendRaw(rawTx) {
    var privateKey = new Buffer(process.env.RINKEBY_KEY, 'hex');
    var transaction = new tx(rawTx);
    transaction.sign(privateKey);
    var serializedTx = transaction.serialize().toString('hex');
    web3.eth.sendSignedTransaction(
    '0x' + serializedTx, function(err, result) {
        if(err) {
            console.log("txn err", err);
        } else {
            console.log("txn result", result);
        }
    });
}
```

Note that this code depends on the RINKEBY_KEY and RINKEBY_ADDRESS environment variables discussed above, so make

sure those are set and available. Once that's done, simply return to your command line and type

node deploy.js

If you have followed all the above faithfully, and if the tutorial gods smile upon you, you will see a result like:

txn count 0
contract deploying...
txn result
0x935c3a2849064d45673b6afd5989e656d5adaff0dc6644b08e33fa2cee57acd9

Congratulations! You have successfully submitted your smart contract to the Rinkeby blockchain. (And unless there's some unforeseen problem it will soon be mined and deployed.)

Let's go over what we just did. First we built a web3 Ethereum client as a Javascript object, and pointed it to the Rinkeby testnet blockchain. Then, in deployContract(), we connected to that blockchain and asked it how many transactions our address had performed (and in passing verified that this connection worked.) We subsequently used that number as the "nonce" for our new transaction, which, to vastly oversimplify, ensures that each transaction is unique.

We then used the "solc" library to compile the Solidity code we previously wrote -- yes, that's right, we ran a Solidity compiler inside of Javascript -- and built a dictionary with all the values required for our new Ethereum transaction. This included the address of our Ethereum account, which we previously set as an environment variable, and the compiled bytecode as its payload. We also set the gasLimit and gasPrice to reasonably generous values, to ensure that we could pay for this transaction out of the money in our account, and finally sent the assembled dictionary off to the "sendRaw" method.

"sendRaw," in turn, converted the dictionary into a transaction object from the "ethereumjs-tx" library, signed it using the private key we created and set as an environment variable above, serialized it, and used our web3 client to actually submit the transaction to the blockchain via a JSON-RPC call over the Internet. Once accepted and mined / paid for, our MyPrize contract will be there on the blockchain

forevermore, its code eternally available for anyone to call.

You can doublecheck the transaction's status by heading over to the Etherscan service for Rinkeby (https://rinkeby.etherscan.io/) and entering into its search box the transaction hash that was logged to your console. You can also use your Ethereum address to get a list of all transactions you've submitted. Don't panic if this transaction doesn't show up immediately -- sometime it can take a minute or so. But it shouldn't take longer than that.

You can get a wealth of in-depth data about your transactions from Etherscan. I encourage you to go through it in some detail. Much may not make sense at first, but, again, you'll wrap your head around it all eventually.

6. Node, Meet Blockchain; Blockchain, Meet Node

Now it's time to move from the command line to the browser. A caveat: as mentioned, I'm generally a Ruby/Rails or Python/Django or Go server guy, so what follows is probably not especially elegant or idiomatic Node code. It should get the job done though.

The principle is simple: get Node to use web3 to connect to our blockchain smart contract, so that anybody in the world can interact with it via our web service, without them having to install a browser plugin or run an Ethereum node. In fact we won't even run a node on our server, though you would if you wanted to scale; instead we'll just keep talking to Rinkeby via Infura.

We're mostly just going to build a JSON API, suitable for use by a smartphone app or a modern Javascript framework like React or Angular. We will, however, also create a very simple HTML page for browser consumption, which (for now) just lets users check on the state of our ten prizes.

We're also going to use Express and Helmet to make Node easier / better, so make sure you run

```
npm install express
```

npm install helmet

before pointing your text editor to "index1.js" within the "node" directory and dumping the following code into it:

```
const express = require('express');
const helmet = require('helmet');

const Web3 = require('web3');
const contractAddress = undefined; // TODO
const abi = [{"constant":false,"inputs":[{"name":"prizeId","type":"uint256"},
{"name":"geohash","type":"bytes10"},
{"name":"_metadata","type":"string"}],"name":"placePrize","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":false,"inputs":[{"name":"prizeId","type":"uint256"},
{"name":"geohash","type":"bytes10"}],"name":"claimPrize","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":
[{"name":"","type":"uint256"}],"name":"prizes","outputs":
[{"name":"id","type":"uint256"},{"name":"owner","type":"address"},
{"name":"geohash","type":"bytes10"},{"name":"metadata","type":"string"},
{"name":"placedAt","type":"uint256"}],"payable":false,"stateMutability":"vie
w","type":"function"},{"inputs":
[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}];

const app = express();
const port = process.env.PORT || 8801;
app.use(helmet());

app.get('/', (req, response) => {
  var html = "<HTML><HEAD><TITLE>Webthereum
Tutorial</TITLE></HEAD><BODY>";
  html += "<FORM action='prize'><SELECT name='prizeId'>";
  for (var i=1; i<=10; i++) {
    html += "<OPTION>"+i;
  }
  html += "</SELECT><INPUT type='submit'></FORM>";
  html += "</BODY></HTML>";
  response.send(html);
});

// Display the prize data, if any
app.get('/prize', (req, response) => {
  get_prize_data(req.query.prizeId, function(err, res) {
    if (err) {
```

```
      response.json( {"err": err} );
    } else {
      response.json( {"res": res} );
    }
  });
});

// error handling: for now just console.log
app.use((err, request, response, next) => {
  console.log(err);
  response.status(500).send('Something broke! '+ JSON.stringify(err));
});

app.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err);
  }
  console.log(`server is listening on ${port}`);
});

// Get prize data
function get_prize_data(prizeId, callback) {
  web3 = new Web3(new
Web3.providers.HttpProvider("https://rinkeby.infura.io/"));
  var contract = new web3.eth.Contract(abi, contractAddress);
  contract.methods.prizes(prizeId).call(function(error, result) {
    if (error) {
      console.log('error', error);
      callback(error, null);
    } else {
      console.log('result', result);
      callback(null, result);
    }
  })
  .catch(function(error) {
    console.log('call error ' + error);
    callback(error, null);
  });
};
```

This time, however, you cannot run the code out of the box. You have to fill in the "contractAddress" and "abi" values at the top first. The contract address is straightforward enough: head to Etherscan, go to the details of the transaction you submitted, and you should see a line to the effect of

[Contract "0x8268205e3e22ccf75615b997ef91e77eeed181aa" Created]

That's your contract address -- ie replace the current line with something like

const contractAddress =
"0x8268205e3e22ccf75615b997ef91e77eeed181aa";

Note that it's a string, i.e. must be between quotes. You could also set and use it as an environment variable, as before, but note that the Ethereum *contract* address is different from your Ethereum *account* address.

You should now just need to run

node index1.js

and point your browser to localhost:8801. A very *very* simple HTML form should appear. Hit "submit" and you should get some raw JSON back, such as --

{"res":
{"0":"1","1":"0xC7679A8C55817EFA649961C6d9DF7596e4bd8C51","2":"0x0
000000000000000000","3":"","4":"0","id":"1","owner":"0xC7679A8C55817E
FA649961C6d9DF7596e4bd8C51","geohash":"0x0000000000000000000","
metadata":"","placedAt":"0"}}

Not super-interesting, yet -- but what's important is that this is real live data coming from your Solidity smart contract, running on the Rinkeby blockchain, to the browser, via your Node service! Give yourself a pat on the back for making this happen; you've earned it.

A Quick Aside

In order to call the contract, your Node code needs to know its interface, aka its ABI, which is hardcoded above. In theory, you should be able to recompile your Solidity contract from source and use its .interface field, rather than hardcoding it. In theory. In practice that seems to not actually work.

So, if you change the MyPrize contract, do the following: Re-open deploy.js. Uncomment the "showABI()" line. Save the file. Then run "node deploy.js" again. The new contract will be deployed, to a new address, and the ABI will be logged to the console. Now, copy-and-paste that ABI back to index1.js; update the contract address (don't use the transaction hash!) in that same file; and you're good to go.

7. End-To-End It, Baby

OK, let's put it all together; let's run a Node web service that you can use to place prizes, to claim prizes, and, what's more, let's add in a Postgres database that you can use to track prize data and transaction information.

Why a database? Because a blockchain is hugely inefficient with respect to both ease-of-development and performance. The idea is to use the blockchain as a source of master truth for the data which needs to be decentralized (such as ownership and location data) and a database as a scalable read cache, and/or for data that doesn't need to be in the blockchain.

Granted, this raises unpleasant issues like "when do you invalidate the cache / update the database?" but such is the price we pay. (One answer which springs to mind: refresh the database every time you write a value to the blockchain, and also via a daily cron job.)

Enough musing; let's code. But first, for the below to work you'll need to

```
npm install pg
npm install express-enforces-ssl
```

-- and, more to the point, get PostgreSQL running locally on your machine, and create a database "webthereum" within it. Explaining Postgres is way out of the scope of this tutorial, I'm afraid, so this section assumes a basic degree of comfort with databases in general and Postgres specifically.

Once you've done that, here is a new and substantially larger file, index.js, for your node directory:

```
// Set up web server
const express = require('express');
const helmet = require('helmet');
const app = express();
const port = process.env.PORT || 8801;
app.use(helmet());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
const express_enforces_ssl = require('express-enforces-ssl');
// TODO: uncomment in production!
// app.enable('trust proxy'); // To force SSL on eg Heroku
// TODO: uncomment in production!
// app.use(express_enforces_ssl());


// Ethereum client setup / data
const contractAddress = undefined; // TODO

const abi = [{"constant":false,"inputs":[{"name":"prizeId","type":"uint256"},
{"name":"geohash","type":"bytes10"},
{"name":"_metadata","type":"string"}],"name":"placePrize","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":false,"inputs":[{"name":"prizeId","type":"uint256"},
{"name":"geohash","type":"bytes10"}],"name":"claimPrize","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":
[{"name":"","type":"uint256"}],"name":"prizes","outputs":
[{"name":"id","type":"uint256"},{"name":"owner","type":"address"},
{"name":"geohash","type":"bytes10"},{"name":"metadata","type":"string"},
{"name":"placedAt","type":"uint256"}],"payable":false,"stateMutability":"vie
w","type":"function"},{"inputs":
[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}];

const Web3 = require('web3');
const web3 = new Web3(new
Web3.providers.HttpProvider("https://rinkeby.infura.io/"));


// Set up database
const { Pool } = require('pg');
const local_db_url = 'postgres://localhost:5432/webthereum';
const db_url = process.env.DATABASE_URL || local_db_url;
```

```
require('pg').defaults.ssl = db_url != local_db_url;
const pool = new Pool({ connectionString: db_url });
pool.on('error', (err, client) => {
  console.error('Unexpected error on idle client', err);
  console.error('client is', client);
  process.exit(-1);
});

const db_creation_string = `
  CREATE TABLE IF NOT EXISTS prizes(id SERIAL PRIMARY KEY, created_at
timestamp with time zone NOT NULL DEFAULT current_timestamp,
updated_at timestamp NOT NULL DEFAULT current_timestamp, latestTxn
CHAR(66), prizeData TEXT);
  CREATE TABLE IF NOT EXISTS transactions(id SERIAL PRIMARY KEY,
created_at timestamp with time zone NOT NULL DEFAULT current_timestamp,
updated_at timestamp NOT NULL DEFAULT current_timestamp, prizeId INT,
txnState INT not null, txnHash CHAR(66) not null, arguments TEXT, txnData
TEXT, txnReceipt TEXT);
  CREATE INDEX IF NOT EXISTS idTxnHash ON transactions(txnHash);
`;


// Enforce HTTPS in non-local environments
app.use((request, response, next) => {
  if (request.hostname == "localhost" || request.hostname == "127.0.0.1" ||
request.secure) {
    next();
  } else {
    response.send("Insecure request on non-localhost server! Uncomment
express_enforces_ssl usage.");
  }
});

// Some minimal error handling
app.use((err, request, response, next) => {
  console.log(err);
  response.status(500).send('Something broke! '+ JSON.stringify(err));
  next();
});


// Create database. Does nothing if already created.
app.get('/create', (req, response) => {
  pool.query(db_creation_string, (err, res) => {
    if (err) {
      response.json( {"err": ""+err} );
```

```
    } else {
      response.send( "Database created: " + res );
    }
  });
});

// Home page
app.get('/', (req, response) => {
  var html = "<HTML><HEAD><TITLE>Webthereum
Tutorial</TITLE></HEAD><BODY>";
  html += "<H3>View</H3>";
  html += "<FORM action='prize'><SELECT name='prizeId'>";
  for (var i=1; i<=10; i++) {
    html += "<OPTION>"+i;
  }
  html += "</SELECT><INPUT type='submit'></FORM>";
  html += "<HR/>";
  html += "<H3>Place</H3>";
  html += "<FORM action='place' method='post'><SELECT
name='prizeId'>";
  for (var j=1; j<=10; j++) {
    html += "<OPTION>"+j;
  }
  html += "</SELECT>";
  html += "<INPUT type='text' name='senderAddress' placeholder='Sender
Address'>";
  html += "<INPUT type='password' name='privateKey' placeholder='Private
Key'>";
  html += "<INPUT type='text' name='geohash' placeholder='Geohash'>";
  html += "<INPUT type='text' name='metadata' placeholder='Metadata'>";
  html += "<INPUT type='submit'></FORM>";
  html += "<HR/>";
  html += "<H3>Claim</H3>";
  html += "<FORM action='claim' method='post'><SELECT
name='prizeId'>";
  for (var k=1; k<=10; k++) {
    html += "<OPTION>"+k;
  }
  html += "</SELECT>";
  html += "<INPUT type='text' name='senderAddress' placeholder='Sender
Address'>";
  html += "<INPUT type='password' name='privateKey' placeholder='Private
Key'>";
  html += "<INPUT type='text' name='geohash' placeholder='Geohash'>";
  html += "<INPUT type='submit'></FORM>";
  html += "<HR/>";
```

```
  html += "<a href='/prizes'>View Current Prize Data Cache</a>";
  html += "</BODY></HTML>";
  response.send(html);
});

// List all the prizes that the database knows about, with their raw JSON data.
app.get('/prizes', (req, response) => {
  var html = "<HTML><HEAD><TITLE>Webthereum
Tutorial</TITLE></HEAD><BODY>";
  html += "<H3>Prizes</H3>";
  html += "<P>This shows the contents of the local database. Actual
blockchain data may now vary.</P>";
  html +=
"<TABLE><TR><TH>ID</TH><TH>Owner</TH><TH>Geohash</TH><TH
>Placed At</TH><TH>Metadata</TH><TH>Latest DB
Txn</TH><TH>State</TH></TR>";
  pool.query("SELECT * FROM Prizes p LEFT OUTER JOIN Transactions t ON
p.latestTxn = t.txnHash ORDER BY p.id", [], (err, res) => {
    if (err) {
      console.log("select prizes error", err);
      html += "Could not get prizes from database";
      response.send(html);
      return;
    }
    for (var i = 0; i < res.rows.length; i++) {
      var row = res.rows[i];
      var prize = JSON.parse(row.prizedata);
      var owner = prize.owner ==
'0x0000000000000000000000000000000000000000' ? "" : prize.owner;
      var txnLink = row.latesttxn === null ? '' : "<a href='/checkTxn?
txnHash=" + row.latesttxn + "'>view</a>";
      var state = res.rows[i].txnstate;
      state = (!state || state == -1) ? "Undefined" : state === 0 ? "Pending" :
state == 1 ? "Mined" : state == 2 ? "Failed" : "Success";
      html += "<TR><TD>" + prize.id + "</TD><TD>" + owner +
"</TD><TD>" + web3.utils.hexToAscii(prize.geohash);
      html += "</TD><TD>" + prize.placedAt + "</TD><TD>" +
prize.metadata + "</TD><TD>" + txnLink + "</TD><TD>" + state +
"</TD></TR>";
    }
    html += "</TABLE></BODY></HTML>";
    response.send(html);
  });
});

// Display the prize data, if any
```

```
app.get('/prize', (req, response) => {
  get_prize_data(req.query.prizeId, function(err, res) {
    if (err) {
      response.json( {"err": err} );
    } else {
      var dbId = req.query.prizeId;
      var dbData = JSON.stringify(res);
      pool.query("INSERT INTO Prizes (id, prizeData) VALUES ($1, $2) ON
CONFLICT (id) DO UPDATE SET prizeData = $2;", [dbId, dbData], (dberr) => {
        if (dberr) {
          response.json( {"err": ""+dberr, "res" : res} );
          return;
        }
        response.json( {"prize" : res} );
      });
    }
  });
});

// Place a prize
app.post('/place', (req, response) => {
  console.log("req", JSON.stringify(req.body));
  place_prize(req.body.prizeId, req.body.geohash, req.body.metadata,
req.body.senderAddress, req.body.privateKey, function(err, txnHash) {
    if (err) {
      response.json( {"err": ""+err, "txnHash" : txnHash} );
      return;
    }
    response.send("Placement submitted: txn "+txnHash);
  });
});

// Register a site as taken, with image URL
app.post('/claim', (req, response) => {
  claim_prize(req.body.prizeId, req.body.geohash, req.body.senderAddress,
req.body.privateKey, function(err, txnHash) {
    if (err) {
      response.json( {"err": ""+err, "txnHash" : txnHash} );
      return;
    }
    response.send("Claim submitted: txn "+txnHash);
  });
});

// Check a transaction
app.get('/checkTxn', (req, response) => {
```

```
  get_eth_txn_receipt(req.query.txnHash, function(err, result) {
    if (result) {
      response.json({"txnHash" : req.query.txnHash, "data" : result});
    } else {
      get_eth_txn_data(req.query.txnHash, function(err2, result) {
        if (result) {
          response.json({"txnHash" : req.query.txnHash, "data" : result});
        } else {
          response.json({"err" : err2, "receiptErr" : err});
        }
      });
    }
  });
});

app.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err);
  }
  console.log(`server is listening on ${port}`);
});


//
// Ethereum calls
//

const STATE_UNDEFINED = -1;
const STATE_PENDING   = 0;
const STATE_MINED     = 1;
const STATE_FAILED    = 2;
const STATE_SUCCESS   = 3;


// Get prize data
var get_prize_data = function(prizeId, callback) {
  var contract = new web3.eth.Contract(abi, contractAddress);
  contract.methods.prizes(prizeId).call(function(error, result) {
    if (error) {
      console.log('error', error);
      callback(error, null);
    } else {
      console.log('result', result);
      callback(null, result);
    }
  })
```

```
  .catch(function(error) {
    console.log('call error ' + error);
    callback(error, null);
  });
};

// Get initial ethereum transaction data
function get_eth_txn_data(txnHash, callback) {
  // do we have the data?
  pool.query("SELECT txnData, txnState FROM Transactions WHERE txnHash
= $1", [txnHash], (err, res) => {
    if (err) {
      console.log("select error", err);
      callback(err, null);
      return;
    }
    if (res.rows.length === 0) {
      console.log("transaction not found", txnHash);
      callback("Transaction not found!", null);
      return;
    }

    var dbState = res.rows[0].txnstate ? res.rows[0].txnstate :
STATE_UNDEFINED;
    var dbData = res.rows[0].txndata ? JSON.parse(res.rows[0].txndata) : null;
    if (dbState >= STATE_MINED && dbData) {
      callback(null, dbData);
      return;
    }

    web3.eth.getTransaction(txnHash).then((txnData) => {
      var txnState = dbState;
      if (dbState <= STATE_PENDING) {
        txnState = txnData.blockNumber === null ? STATE_PENDING :
STATE_MINED;
      }
      pool.query("UPDATE Transactions SET txnData = $1, txnState = $2
WHERE txnHash = $3", [JSON.stringify(txnData), txnState, txnHash], (err) =>
{
        if (err) {
          console.log("update error", err);
          callback(err, txnData);
        } else {
          callback(null, txnData);
        }
      });
```

```
    }).catch((error) => {
      console.log("web3 txn data error", error);
      callback(error, null);
    });
  });
}

// Get processed ethereum transaction receipt
function get_eth_txn_receipt(txnHash, callback) {
  // do we have the receipt?
  pool.query("SELECT txnData, txnReceipt, txnState FROM Transactions
WHERE txnHash = $1", [txnHash], (err, res) => {
    if (err) {
      console.log("select err", err);
      callback(err, null);
      return;
    }
    if (res.rows.length === 0) {
      console.log("txn not found", txnHash);
      callback("Transaction not found!", null);
      return;
    }

    var dbReceipt = res.rows[0].txnreceipt ?
JSON.parse(res.rows[0].txnreceipt) : null;
    if (dbReceipt) {
      var dbData = res.rows[0].txndata ? JSON.parse(res.rows[0].txndata) :
null;
      var dbState = res.rows[0].txnstate ? res.rows[0].txnstate :
STATE_UNDEFINED;
      callback(null, { "state" : dbState, "receipt" : dbReceipt, "initialData" :
dbData });
      return;
    }

    web3.eth.getTransactionReceipt(txnHash).then((receipt) => {
      var txnState = receipt.status === "0x1" ? STATE_SUCCESS :
STATE_FAILED;
      pool.query("UPDATE Transactions SET txnReceipt = $1, txnState = $2
WHERE txnHash = $3", [JSON.stringify(receipt), txnState, txnHash], (err) =>
{
        if (err) {
          console.log("update err", err);
          callback(err, receipt);
        } else {
          callback(null, {"receipt" : receipt});
```

```
      }
    });
  }).catch((error) => {
    console.log("web3 receipt error", error);
    callback(error, null);
  });
 });
}

//
// Ethereum transactions
//

const GAS_LIMIT = 4000000; // should not be a constant if using real money
const GAS_PRICE = 20000000000; // should not be a constant if using real
money

// Place a prize
function place_prize(prizeId, geohash, metadata, senderAddress, privateKey,
callback) {
 web3.eth.getTransactionCount(senderAddress).then((txnCount) => {
  console.log("prizeId", prizeId);
  var contract = new web3.eth.Contract(abi, contractAddress);
  var geohashBytes = web3.utils.asciiToHex(geohash);
  var placeMethod = contract.methods.placePrize(prizeId, geohashBytes,
metadata);
  var encodedABI = placeMethod.encodeABI();
  var placeTx = {
   from: senderAddress,
   to: contractAddress,
   nonce: web3.utils.toHex(txnCount),
   gasLimit: web3.utils.toHex(GAS_LIMIT),
   gasPrice: web3.utils.toHex(GAS_PRICE),
   data: encodedABI,
  };
  sendTxn(privateKey, placeTx, prizeId, {"args" : { "geohash" : geohash,
"metadata" : metadata } }, callback );
 }).catch((err) => {
  console.log("web3 err", err);
  callback(err, null);
 });
}

// Claim a prize
function claim_prize(prizeId, geohash, senderAddress, privateKey, callback) {
 web3.eth.getTransactionCount(senderAddress).then((txnCount) => {
```

```
    var contract = new web3.eth.Contract(abi, contractAddress);
    var geohashBytes = web3.utils.asciiToHex(geohash);
    var claimMethod = contract.methods.claimPrize(prizeId, geohashBytes);
    var encodedABI = claimMethod.encodeABI();
    var claimTx = {
      from: senderAddress,
      to: contractAddress,
      nonce: web3.utils.toHex(txnCount),
      gasLimit: web3.utils.toHex(GAS_LIMIT),
      gasPrice: web3.utils.toHex(GAS_PRICE),
      data: encodedABI,
    };
    sendTxn(privateKey, claimTx, prizeId, {"args" : { "geohash" : geohash } },
callback );
  }).catch((err) => {
    console.log("web3 err", err);
    callback(err, null);
  });
}

function sendTxn(privateKey, rawTx, prizeId, args, callback) {
  var tx = require('ethereumjs-tx');
  var privateKeyBuffer = new Buffer(privateKey, 'hex');
  var transaction = new tx(rawTx);
  transaction.sign(privateKeyBuffer);
  var serializedTx = transaction.serialize().toString('hex');
  web3.eth.sendSignedTransaction(
  '0x' + serializedTx, function(err, txnHash) {
    if(err) {
      console.log("txn err", err);
      callback(err, null);
    } else {
      console.log("txn result", txnHash);
      pool.query("INSERT INTO Transactions (prizeId, txnHash, txnState,
arguments) VALUES ($1, $2, $3, $4)", [prizeId, txnHash, STATE_PENDING,
args], (err) => {
        if (err) {
          console.log("insert err", err);
          callback(err, txnHash);
        } else {
          pool.query("INSERT INTO Prizes (id, latestTxn) VALUES ($1, $2) ON
CONFLICT (id) DO UPDATE SET latestTxn = $2;", [prizeId, txnHash], (dberr)
=> {
            if (dberr) {
              callback(dberr, null);
            } else {
```

```
            callback(null, txnHash);
          }
        });
      }
    });
  }
 }).catch((err) => {
  callback(err, null);
 });
}
```

Save that. Replace the "contractAddress = undefined" with your real address, as before. Then fire up

node index.js

and point your browser at localhost:8801, and you should see a slightly more full-featured (if extremely spartan and ugly) web page -- one that lets you not just view prize data, but also place or claim prizes. You can also view the data stored in the local database, and check the details of what the database thinks is the most recent blockchain transaction relating to that prize.

Note however that this is highly asynchronous. When you "place" or "claim" a prize, you're really just submitting a transaction to the blockchain; it takes a minute or so, an eternity in web time, for that transaction to be accepted or rejected. (It might be rejected because e.g. there isn't enough ether in the sender's account to pay for it.) In a real web app this delay must be handled by the server code, and communicated to the user, in an elegant manner.

Otherwise there isn't a lot of conceptually new code here. As with the previous section, you use the Node service as an intermediary between the blockchain and the browser. As with deploying the contract in the first place, to write data to the blockchain you build Ethereum transactions and sign them. The big difference here is that we aren't restricted to a single address stored in an environment variable; we let *anyone* with a blockchain address and private key perform these transactions.

On the one hand this approach arguably combines the decentralized power of blockchain apps with the accessibility and scalability of web

applications. On the other, though --

Security and the Browser <> Server <> Blockchain Pipeline

Private keys are called that for a reason. Ethereum addresses can hold very, very significant amounts of money, and their private keys should be guarded accordingly. This means that as a general rule you should not ever be pasting them into a browser. You face potential client-side plugin attacks, Man-in-the-Middle attacks, cross-site scripting attacks, server-side hack attacks, etc etc etc.

It's fine for this tutorial which uses fake money, of course. (Though even this tutorial makes a conscious point of enforcing HTTPS if the server is not running locally, using a password HTML field for the private key, etc.) But if you want to build a real blockchain-powered web service, you're either going to have to be smarter about security than this, or you'll have to outsource being smarter to your users, e.g. by telling them to only use throwaway, low-value Ethereum addresses for your service.

That last may sound callous and insecure, and it is, but in the long run maybe part of the correct security solution is for users to have "saving" Ethereum addresses, with real money, which they keep very secure, from which they occasionally transfer money to "spending" addresses which they actually use for application transactions.

Note: *part of*. Even then you shouldn't be pasting private keys into web forms. Perhaps private keys should be decomposed into two halves, unique to a given service, when a user registers: the server maintains one half, the user enters the other half (which the server never stores) to validate each transaction, and if either gets hacked, well, each half is useless without the other. Perhaps some centralized (or even decentralized) private-key manager such as 1Password or LastPass will arise to solve this problem in general.

Regardless, the thing to remember is that the considerable value of ether currency, combined with the irrevocable nature of Ethereum transactions, means that if you're building a *real* Ethereum web app, you have to take private-key security very seriously. You have been

warned.

That's all, folks!

Thus endeth this tutorial. Your thoughts and feedback are (probably) welcome, so feel free to email me or @ me on Twitter; details below. Thanks in advance --

Jon Evans
jon@happyfuncorp.com
twitter.com/rezendi