

← Back

How To Learn Solidity: The Ultimate Ethereum Coding Guide

An in-depth guide by BlockGeeks

9

(<https://blockgeeks.com/guides/solidity/#comments>)

Solidity (<https://blockgeeks.com/category/solidity/>)

7

661

328

392

1K SHARES

Join over 115,115 Members
Angel Investors, Startups & Blockchain developers...

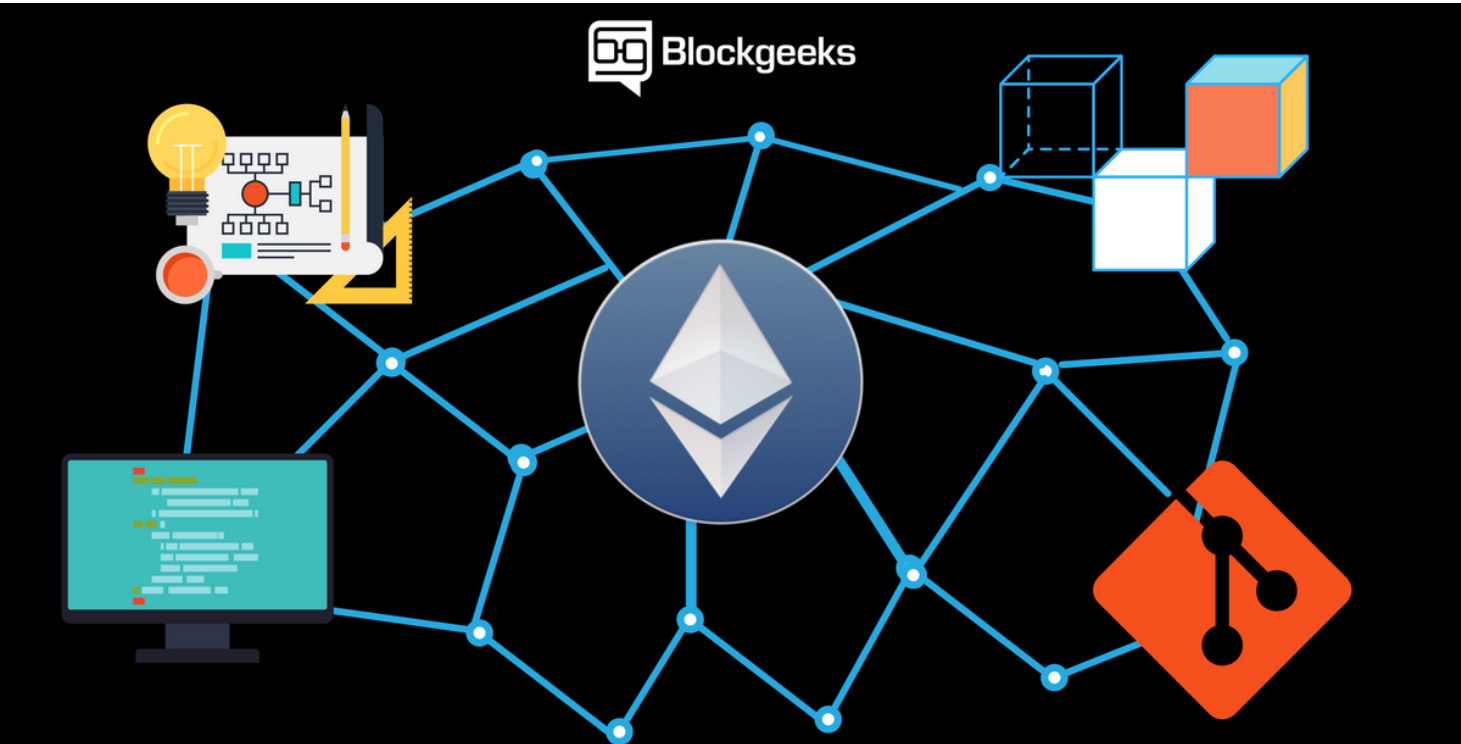
Enter your email address

Yes, Get Access!

No Thanks!

This Guide will walk you step -by-step in learning Solidity.

The Ethereum Foundation (<https://blockgeeks.com/guides/what-is-ethereum/>) has been shaking up the world of blockchain since the early days of the project, around late 2013 and early 2014. Ethereum really kickstarted the “Bitcoin 2.0” and what we think of as the “blockchain (<https://blockgeeks.com/guides/what-is-blockchain-technology/>)” movement, after the first big Bitcoin (<https://blockgeeks.com/guides/what-is-bitcoin-a-step-by-step-guide/>) “bubble” up past \$1000 USD on the markets got everyone’s attention. Ethereum is a blockchain project with a cryptocurrency, Ether, similar to Bitcoin, but Ethereum has the added feature of a (nearly) Turing- complete virtual machine language and processing capability embedded into the node implementation.



The Ethereum Virtual Machine (EVM) allows Ethereum nodes to actually store and process data in exchange for payment, responding to real-world events and allowing a lot of new opportunities to support on-chain applications that were never before available to developers and real-world users. I had the luck to actually be in Switzerland in early 2014, and to get to visit an Ethereum “holon” and hang out with some of the Ethereum founders before the Ether token sale, back when they were “self-funded”.

I asked Mihai Alisie what an Ethereum smart contract is, and he explained:

“Smart-contracts are a way for people all across the globe to do business with each other even if they don’t speak the same language or use the same currency.”

So that’s really the perspective I begin with, the idea that we can define programmatically the rules of a business contract, in a simple machine language, to bring people together and allow them to conduct business in a trustable, secure, and automated fashion.

Solidity Language itself is a tool that we use to generate machine-level code that can execute on the EVM, it’s a language with a *compiler* which takes our high-level human-readable code and breaks it down into simple instructions like “put data into a register”, “add data from two registers”, “jump back to instruction at memory point xxxxx”, which form the basis of any microprocessor executable program.

How To Create An Ethereum Smart Contract



So the Solidity language is just one of several languages which can be compiled into EVM bytecode, another language that does the same thing is called Serpent. Each language might have several compiler tools but they all do the same thing, which is to generate EVM machine-level bytecode to be run on the Ethereum nodes, for payment.

How To Learn Solidity

Solidity itself is a pretty simple language, as far as programming languages go.

In fact, it is a *purposefully slimmed down*, loosely-typed language with a syntax very similar to ECMAScript (Javascript). There are some key points to remember from the Ethereum Design Rationale (<https://github.com/ethereum/wiki/wiki/Design-Rationale>) document, namely that we are working within a stack-and-memory model with a 32-byte instruction word size, the EVM gives us access to the program “**stack**” which is like a register space where we can also stick memory addresses to make the Program Counter loop/jump (for sequential program control), an expandable temporary “**memory**” and a more permanent “**storage**” which is actually written into the permanent blockchain, and most importantly, the EVM requires total **determinism** within the smart contracts.

This requirement for determinism is the reason you won’t see the “random()” function in Solidity language. When an ethereum block is “mined”, the smart-contract (<https://blockgeeks.com/guides/smart-contracts/>) deployments and function calls within that block (meaning those lined up to happen within the last block duration) get executed on the node that mines the block, and the new state changes to any storage spaces or transactions within that smart-contract actually occur on that miner node. Then the new block gets propagated out to all the other nodes and each node tries to independently verify the block, which includes doing those same state changes to their local copy of the blockchain also. Here’s where it will fail if the smart-contract (<https://blockgeeks.com/guides/smart-contracts/>) acts non-deterministically. If the other nodes cannot come to a consensus about the state of blockchain after the new block and its contracts get executed, the network could literally halt.

This is why Ethereum smart-contracts (and smart-contracts in general in any blockchain system) must be deterministic: so that the network of nodes can always validate and maintain consensus about the new blocks coming in, in order to continue running.

How to Write an Ethereum Election Smart Contract



Another limitation you'll find in EVM smart-contracts is the inability to access data outside the "memory", and "storage", (we don't want the smart-contract to be able to read or delete the hard-drives of the nodes it runs on), and the inability to query outside resources like with a JQuery. We don't really have access to many library functions like for parsing JSON structures or doing floating-point arithmetic, and it's actually cost-prohibitive to do those sub-routines or store much data in the ethereum blockchain itself.

When you call a smart-contract that does some state-changing work or computation (any action besides other than simply reading from storage), you will incur a **gas "cost"** for the work done by the smart contract, and this gas cost is related to the amount of computational work required to execute your function. It's sort of a "micropayment for microcomputing" system, where you can expect to pay a set amount of gas for a set amount of computation, forever.

The price of gas itself is meant to stay generally constant, meaning that when Ether goes up on the global markets, the **price** of gas against Ether should go down. Thus, when you execute a function call to a smart-contract, you can get an estimation of the amount of gas you must pay beforehand, but you must also specify the **price** (in ether per gas) that you are willing to pay, and the mining nodes can decide if that's a good enough rate for them to pick up your smart-contract function call in their next block.

Smart-contracts have their own address, from which they can receive and send Ether. Smart contracts can track the "caller" of the function in a verifiable way, so it can determine if one of its functions is being called by a privileged "owner" or "admin" account, and act accordingly for administrative functions. They have the ability to read data from the Ethereum blockchain, and access info on transactions in older blocks. But are smart-contracts "locked in" into their own little deterministic world, only able to be aware of data stored in the Ethereum blockchain itself?

Not at all! That's where "**oracles**" come in. We can make a call to an oracle that will tell us something about the outside world in a trustable way, and *act on that data* within the smart contract. The key point here is that even though real-world events themselves are not deterministic, the Oracle can be trusted to always answer every node's request about what happened in a deterministic way, so that all nodes can still come to a consensus. So this is how the trusted Oracle works in theory: an "oracle" will take some data, say a ticker price feed about a real-world stock price, and record that data into "storage" in a simple Oracle smart-contract, something like this

(see also the explanation here (<https://ethereum.stackexchange.com/questions/11589/how-do-oracle-services-work-under-the-hood>)):

```
function storeTickerData(bytes8 symbol, uint open, uint high, uint low, uint close)
    onlyOracle
    returns(bool success)
{
    # store ticker data on-chain
    tickerData[symbol][block.number] = [open,high,low,close]
    # tickerData is a map of maps, string => uint => array[uint,uint,uint,uint]
    return true;
}
```

There are companies that specialise in being the trusted oracle, and designing systems to disintermediate themselves from having to be trusted with the datafeed. If we take a look at the Oraclize documentation (<https://docs.oraclize.it/>) we see this interesting quote:

In particular, the aim is not to force smart contract developers in having to trust Oraclize with the data they need. Without any backing of authenticity, Oraclize could easily tamper with the data. This is why, in order to complete this complex task, Oraclize returns the data requested along with a proof of the authenticity: i.e that the data comes from the data provider which has been explicitly demanded by

the smart contract.

So you should check out that docs section on “authenticity proofs” for more details, but the idea is that even when new Ethereum nerds, at some distant time in the future, spin up their Ethereum nodes and start to sync their blockchains to the network, all the data they need to execute all those “oracle-dependent” smart-contracts is safely on-chain for them to download, forever. The matter of who pays the “gas” for the oracle to do all this data-writing about real-world events (blockchain storage “state-changing” costs gas) is another topic, but basically, it gets prepaid by the data requester in one of several ways. The cool thing here is that even though we can’t write a smart-contract that “throws dice” in the sense of including actual randomness, it *is* actually possible to write a smart-contract that reacts to the throw of a die, as officiated by an oracle.

Solidity Basics

Basic Code Background and Setup

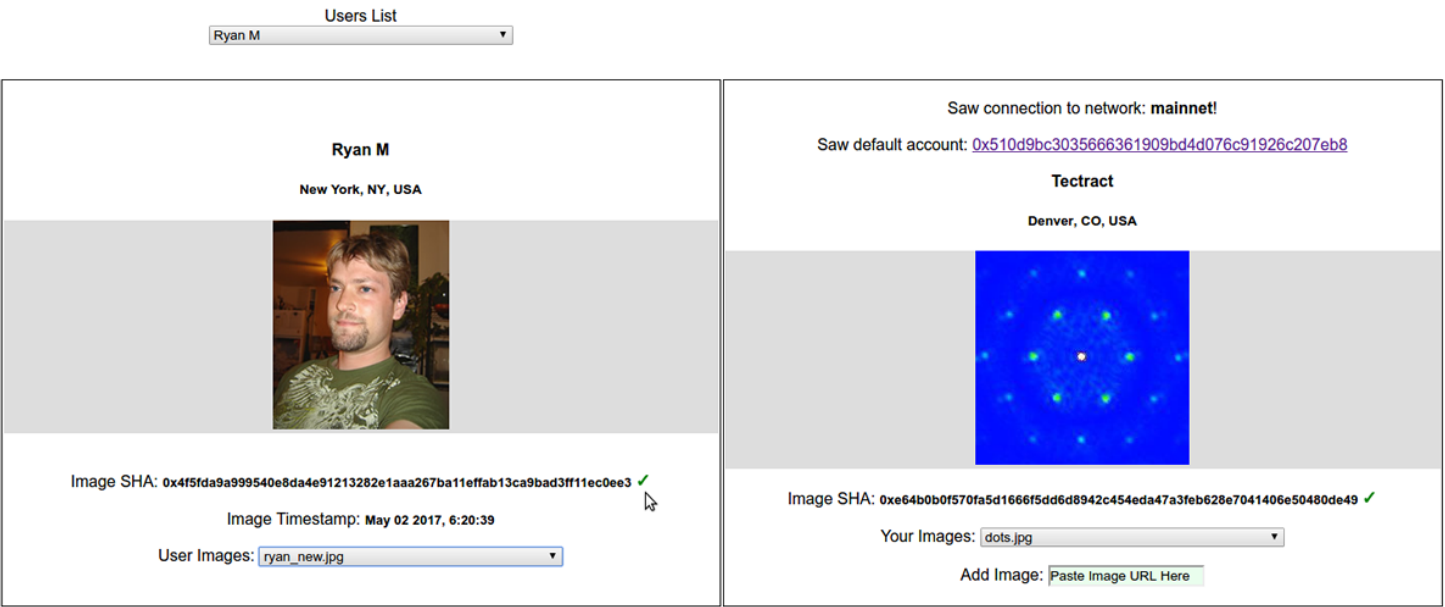
So let’s get started with a basic Ethereum smart-contract, written in Solidity language. The best way to learn is by doing! We can see from the little snippet of code above that we have a pretty simple function declaration syntax, and some basic data types (<http://solidity.readthedocs.io/en/develop/types.html>) like ‘uint’ (unsigned integer) and ‘string’, and we see in the docs we also have variable types like ‘address’, which contains ‘.balance’ and ‘.transfer’ methods, which predictably return the balance at an Ethereum address, and allow the smart-contract to actually send Ether to that address, with units denominated in Wei (https://en.bitcoin.it/wiki/Wei_Dai).

We have your basic arrays, enums, operators, hash data structures called ‘mappings’, and some special pre-defined units and global vars (<http://solidity.readthedocs.io/en/develop/units-and-global-variables.html>) which include things like blockheight, last block timestamp and some handy SHA-hash and address/key manipulation functions. One important quirk about Solidity when working with multi-dimensional arrays: indexing is in “reversed” order from most languages *during declaration only*, as noted in the docs:

An array of fixed size k and element type T is written as T[k], an array of dynamic size as T[]. As an example, an array of 5 dynamic arrays of uint is uint[][5] (note that the notation is reversed when compared to some other languages). To access the second uint in the third dynamic array, you use x[2][1] (indices are zero-based and access works in the opposite way of the declaration...)

Ok, we have enough basic tools here to be dangerous, let’s get started. Let’s do some sort of basic social app where people can have some images and some basic info about themselves posted in the blockchain. Each user will have an ethereum address and a “handle” or username associated with their account. We’ll have some basic demographic data that people may or may not enter, and then we’ll have a simple search page where you can look up users in a basic directory, and check out the images they have added/notarized to their user profile.

Here’s an image of what our final Ethereum Solidity smart-contract Decentralized App (Dapp) User Interface product will look like, with users and notarized images added:



I’ve notarized a nice picture of myself for you guys, and an image of a Galium Arsenide Crystal structure from my grad school days for you. You can see the SH256 notary hashes and timestamps for each image and a little checkmark indicating that the notarized image data still matches the notary hash that was created at the indicated timestamp. So I’m going to walk you through how I built this app, pointing out the interesting tidbits for you, with all the links and code examples you might need for future reference. Sound good? Let’s go!

for you, with all the links and code examples you might need for future reference. Sound good? Let's go!

The first thing I'll do is set up a Truffle (<https://github.com/trufflesuite/truffle>) "test-bed" that will allow me to easily test compiling my smart contract. Compiling and deploying smart-contracts involves generating long signed transactions, but Truffle will allow me to compile/deploy and test my smart-contract using simple commands. There are a bunch of tools that can do this, but I happen to like this Solidity smart-contract video

tutorial (<https://www.youtube.com/watch?v=3-XPBtAfcqo>) which uses the Truffle (<https://github.com/trufflesuite/truffle>) tool. I have a demo github project (<https://github.com/Tectract/ethereum-demo-tools>) folder I've created just for this tutorial for everyone to follow along. I'll start with some simple setup commands (under a Debian linux environment):

```
tectract@blockgeeks/ethereum-demo-tools> npm i -g truffle
# if you run into permissions error globally installing node modules, see here

tectract@blockgeeks/ethereum-demo-tools> mkdir Geekt # I'm going to call my social application "Geekt"
tectract@blockgeeks/ethereum-demo-tools> cd Geekt
tectract@blockgeeks/ethereum-demo-tools/Geekt> truffle init # this creates a few folders and setup files, within the Ge
tectract@blockgeeks/ethereum-demo-tools/Geekt > ls
contracts migrations test truffle.js          # we have created three folders, and one new file called truff

tectract@blockgeeks/ethereum-demo-tools/Geekt > ls contracts/
ConvertLib.sol MetaCoin.sol Migrations.sol      # we have some basic template contracts generated by Truffle

tectract@blockgeeks/ethereum-demo-tools/Geekt > ls migrations/
1_initial_migration.js 2_deploy_contracts.js    # a couple files for compilation/migration settings with Truff
```

A Simple Solidity Smart-Contract Example

Ok, I'm going to create a contract called Geekt.sol, in the /Geekt/contracts folder that was generated when I called 'truffle init'. You can see the full contract code here (<https://github.com/Tectract/ethereum-demo-tools/blob/master/GeektSolidity/contracts/Geekt.sol>), and I'll point out the most interesting bits here. At the top of the file we a line that specifies the compiler version and some basic contract definition syntax and variable definitions.

```
pragma solidity ^0.4.4;
contract Geekt {

    address GeektAdmin;

    mapping ( bytes32 => notarizedImage) notarizedImages; // this allows to look up notarizedImages by their SHA256notary
    bytes32[] imagesByNotaryHash; // this is like a whitepages of all images, by SHA256notaryHash

    mapping ( address => User ) Users; // this allows to look up Users by their ethereum address
    address[] usersByAddress; // this is like a whitepages of all users, by ethereum address
```

When you call a state-changing function in the real world, you will have to specify how much ether you are will to pay in order for that, but luckily you can request an estimate of the cost in terms of gas cost and gas price (in gas/ether) beforehand, and it's usually very accurate. The Ethereum miners will decide if you paid enough, and include your state-changing transaction in the next block, so you actually have to wait for those functions to return when the next block is found. Reading data out of these "storage" structures is free and you don't have to wait for the next block either, nodes will just read the data and return it to you right away.

The Ethereum miners will decide if you paid enough, and include your state-changing transaction in the next block, so you actually have to wait for those functions to return when the next block is found. Reading data out of these "storage" structures is free and you don't have to wait for the next block either, nodes will just read the data and return it to you right away.

So the **Users** mapping is our main storage object that allows us to create User objects and look them up by address. Mappings are super-efficient for storing data and retrieving it quickly, but there's no easy way to iterate a map, so you can see I also created a **usersByAddress** address array that holds every known address of a user in our system, like a white-pages of all users. I created the

I created the **notarizedImages** mapping to allow us to create "image" objects in storage and look them up by an associated SHA256 hash of the image data, and those notaryHash indexes are 32-bytes long.

We also have an **ImagesByNotaryHash** bytes32 array which is a list of all notaryHashes, like a white-pages allowing us to iterate all the images that have been notarized.

```
struct notarizedImage {
    string imageURL;
    uint timestamp;
```

```

    uint timestamp;
}

struct User {
    string handle;
    bytes32 city;
    bytes32 state;
    bytes32 country;
    bytes32[] myImages;
}

```

These (above) are very basic structs, and they also represent “storage” objects, so they cost real ether to “create” and the data in them is actually stored in the Ethereum blockchain, and it costs real ether to change the “state” of their internal variables. We really just used the outer mappings and arrays to keep track of where in the blockchain memory that these image/user structures reside. Our **notarizedImage** struct stores simply a URL to an image, somewhere on the web presumably, and a timestamp notifying when the image was notarized. Our **User** struct stores some basic user: handle, city, state, country, and it also stores an array of all the images that this user has notarized and “added” to their user account, like a mini user-by-user version of the global **imagesByNotaryHash** white-pages object.

```

function registerNewUser(string handle, bytes32 city, bytes32 state, bytes32 country) returns (bool success) {
    address thisNewAddress = msg.sender;
    // don't overwrite existing entries, and make sure handle isn't null
    if(bytes(Users[msg.sender].handle).length == 0 && bytes(handle).length != 0){
        Users[thisNewAddress].handle = handle;
        Users[thisNewAddress].city = city;
        Users[thisNewAddress].state = state;
        Users[thisNewAddress].country = country;
        usersByAddress.push(thisNewAddress); // adds an entry for this user to the user 'whitepages'
        return true;
    } else {
        return false; // either handle was null, or a user with this handle already existed
    }
}

```

Here’s our **registerNewUser** function. It takes handle, city, state, country as input variables and returns true or false to indicate success or failure. Why would it fail? Well, we don’t want to allow users to over-write each other’s handle, so this is going to be like a first-claim system for handles. If a user with this handle already exists, we return null, so I have placed a conditional “if” line there to check for that. We also don’t allow the creation of a username with no handle. One thing we note is the **thisNewAddress** which is the caller of the function, you can see we use the special *msg.sender* function to grab that data, this is the address that “do this function” transaction was sent from when someone (anyone) calls our smart-contract function, and they pay real ether to do it. So a “mapping” object is null by default, and when we call:

```
Users[thisNewAddress].handle = handle;
```

This creates a new User object, in our Users mapping, and sets the handle. Same with city, state, country. Note we also push the the new User’s address into our usersByAddress global Users “white-pages” object before we return true there.

```

function addImageToUser(string imageURL, bytes32 SHA256notaryHash) returns (bool success) {
    address thisNewAddress = msg.sender;
    if(bytes(Users[thisNewAddress].handle).length != 0){ // make sure this user has created an account first
        if(bytes(imageURL).length != 0){ // ) { // couldn't get bytes32 null check to work, oh well!
            // prevent users from fighting over sha->image listings in the whitepages, but still allow them to add a personal
            if(bytes(notarizedImages[SHA256notaryHash].imageURL).length == 0) {
                imagesByNotaryHash.push(SHA256notaryHash); // adds entry for this image to our image whitepages
            }
            notarizedImages[SHA256notaryHash].imageURL = imageURL;
            notarizedImages[SHA256notaryHash].timeStamp = block.timestamp; // note that updating an image also updates the timestamp
            Users[thisNewAddress].myImages.push(SHA256notaryHash); // add the image hash to this users .myImages array
            return true;
        } else {
            return false; // either imageURL or SHA256notaryHash was null, couldn't store image
        }
    }
    return true;
} else {
    return false; // user didn't have an account yet, couldn't store image
}
}

```

Our **addImageToUser** to user function is pretty similar to the **registerNewUser** function. It looks you up by your sending address via the special *msg.sender* object and checks that there is a registered User (by handle) for that address before it tries to add an image for your user entry. We allow users to add their notaryHash to the global white-pages of images only if it doesn’t exist, to prevent users from fighting over entries in the global white-pages, but we allow them to add/update any notaryHash entry within their own mini-whitepages of their own images. Nothing to it!

```
function getUsers() constant returns (address[]) { return usersByAddress; }

function getUser(address userAddress) constant returns (string,bytes32,bytes32,bytes32,bytes32[]) {
    return (Users[userAddress].handle,Users[userAddress].city,Users[userAddress].state,Users[userAddress].country,Users[userAddress].myImages);
}

function getAllImages() constant returns (bytes32[]) { return imagesByNotaryHash; }

function getUserImages(address userAddress) constant returns (bytes32[]) { return Users[userAddress].myImages; }

function getImage(bytes32 SHA256notaryHash) constant returns (string,uint) {
    return (notarizedImages[SHA256notaryHash].imageUrl,notarizedImages[SHA256notaryHash].timeStamp);
}
}
```

Finally, the (above) accessor functions allow us to simply read out each user or image, or get the full white-pages listings of all users or all images. Note these functions return **constant** data, which basically means they are read-only, not “state-changing” and they are free to call and they return the data right away, you don’t have to wait until the next block. We basically just call into our global mappings / arrays and return the relevant data: users by address or images by notaryHash.

Compiling & Testing a Smart-Contract

Now we want to actually test our smart-contract locally, in a test environment. It’s really easy using simple Truffle commands, but first we actually need a **local Ethereum node** to test against. That’s where Ethereum TestRPC (<https://github.com/ethereumjs/testrpc>) comes in. TestRPC is basically a fake node, a slim program that just pretends to be a node and responds like a node would respond on your localhost machine. TestRPC runs on port 8545 like a normal Ethereum node, and it has the ability to compile Solidity smart-contracts into EVM code and run that code too, plus you get instant responses for testing, instead of having to wait on the real Ethereum network to find the next block. Now, you could run Truffle test compile/deploy commands against a real Ethereum node, but that would cost real Ether, plus it’s time-consuming and memory consuming to run your own node if you don’t need to. So we do a quick few install commands:

```
npm i -g ethereum-testrpc
testrpc -m “sample dog come year spray crawl learn general detect silver jelly pilot”
```

This starts up TestRPC with a specified “seed” phrase, and you should see something like this as the output:

```
EthereumJS TestRPC v3.0.5

Available Accounts
=====
(0) 0x0ac21f1a6fe22241ccd3af85477e5358ac5847c2
(1) 0xb0a36610de0912f2ee794d7f326acc4b3d4bc7bc
...
(9) 0x4c1cc45ef231158947639c1eabec5c5cb187401c

Private Keys
=====
(0) 91e639bd434790e1d4dc4dca95311375007617df501e8c9c250e6a001689f2c7
(1) afaeff0fc68439c4057b09ef1807aaf4e695294db57bd631ce0ddd2e8332eea7
...
(9) dcc51540372fa2cf808efd322c5e158ad5b0dbf330a809c79b540f553c6243d7

HD Wallet
=====
Mnemonic:      sample dog come year spray crawl learn general detect silver jelly pilot
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

You’ll see some activity here in this window with TestRPC running, as you deploy and interact with your smart-contract via Truffle or Web3.js. Ok, we’re ready for test deployment. You’ll need to modify the file `/migrations/2_deploy_contracts.js` to include the name of your smart-contract, for truffle to know to compile and deploy it. We do this command:

```
truffle compile
```

If all goes well, you’ll see a message saying it’s “saving artifacts”, and no error messages. I’m not going to lie, if your contract has syntax errors or other issues, the compiler error messages you’ll see will likely be mysterious and hard to interpret! If you get an error about the “stack size” that

Other issues, the compiler error messages you see will likely be mysterious and hard to interpret. If you get an error about the "stack size", that probably means you have too many variables being passed into/out of a function call, a good rule to remember is 16 variables input or output is the max, in general. Also, remember that Solidity smart-contracts cannot return custom struct data types, so you'll have to work around that too, usually I just return arrays and pointer/addresses of other structs in my internal mappings. If you get *runtime* errors message about the "stack", that may mean you have a bad conditional in your code.

truffle migrate

This command (above) actually does the test deployment of your smart-contract onto your TestRPC node. I see this output return:

```
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0xd06a1935230c5bae8c7ecf75fbf4f17a04564ed8
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Geekt...
  Geekt: 0xe70ff0fa937a25d5dd4172318fa1593baba5a027
Saving successful migration to network...
Saving artifacts...
```

Our smart-contract (named “Geek”) is given an address on the Ethereum blockchain when it deploys successfully, you can see above, the address is **0xe70ff0fa937a25d5dd4172318fa1593baba5a027**. On a real, live Ethereum network, you pay gas to deploy your contract and the address never changes. On TestRPC, if you shut down TestRPC it will forget everything, and once you start TestRPC back up again, you’ll have to re-deploy your contract and you’ll get a different address for your smart-contract. The address of your smart-contract is where people can send transactions with messages to interact with it, to do state-changing transactions or just read data out of the Ethereum blockchain. Smart-contracts can also interact *directly with each other* via these addresses, using “messages”. Smart-contracts that interact with other smart-contracts to store, change, or read data via these Ethereum blockchain “messages” are known as **Decentralized Autonomous Organizations**, or DAO’s.

Ok, now comes the fun part. We are ready to actually do some initial testing and interact with our smart-contract. We start up the Truffle “console” and do some queries against our TestRPC localhost node, to make sure everything is working and we can add users and images and retrieve them properly.

```
truffle console
> Geekt = Geekt.deployed()
> Geekt.then(function(instance){return JSON.stringify(instance.abi);})
> Geekt.then(function(instance){return instance.registerNewUser("Tectract","Denver","CO","USA");})

> Geekt.then(function(instance){return instance.addImageToUser('www.myimageURL.com','0x6c3e007e281f6948b37c511a11e43c80...');})

> Geekt.then(function(instance){return instance.getUser('0x0ac21f1a6fe22241ccd3af85477e5358ac5847c2');}) > Geekt.then(f...
> Geekt.then(function(instance){return instance.getImages();})
> Geekt.then(function(instance){return instance.getImage('0x6c3e007e281f6948b37c511a11e43c8026d2a16a8a45fed4e83379b66b0...');
```

One important concept here is the ABI, which is a javascript-formatted object that describes the function names and inputs/outputs for interacting with your smart-contract, it's sort of like the Application Programmer Interface (API) description for your smart-contract, that tells people how to form messages for it. My `registerNewUser()` function is working! I see this in response to the `registerNewUser()` function call, in the console window:

```
{ tx: '0x1b9f55971871921ccd23a9aa7620620c6c958a893af334087283926d4c6d60b1',
  receipt:
    { transactionHash: '0x1b9f55971871921ccd23a9aa7620620c6c958a893af334087283926d4c6d60b1',
      transactionIndex: 0,
      blockHash: '0x2be4fab68daaf8db199e2a6adea101c0f1ed06f46aba21e8e4c06e752ca3325c',
      blockNumber: 5,
      gasUsed: 145215,
      cumulativeGasUsed: 145215,
      contractAddress: null,
      logs: [] },
  logs: [] }
```

AddImageToUser() function returns success similarly, and when I can now retrieve individual user records or notarized image records from the Ethereum blockchain. My getUser() function call returns:

[illegible]

This all looks great. Everything is working, tested in TestRPC and Truffle, and we're ready to actually build a cool graphical user interface for our Dapp so the world can interact with our user registration and image notary on the Ethereum blockchain.

Basic Ethereum DApp Design

Localhost Design & Testing

We're ready to go, our Solidity-language smart-contract has been successfully compiled, and deployed to our local testRPC Ethereum node. Now we can quickly build a simple DApp that can allow users to interact with the smart-contract through their web-browser, using some basic web programming and the special Web3.js (<https://github.com/ethereum/web3.js/>) javascript module, made specifically to interact with Ethereum nodes and smart-contracts over the web.

I have made this demo (<https://github.com/Tectract/ethereum-demo-tools/tree/master/GeektReactApp>) (open-source github repositories) as a reference for people to use while building Ethereum DApps using web3.js, This demo was made with an easy-to-use tool called Create-React-App (<https://github.com/facebookincubator/create-react-app>), it uses a the Facebook-internal web-language React, but we won't be focusing on any React code, just the web.js javascript commands needed. My code is in CommonJS, it's almost exactly the same as plain-old ES6 javascript.

So in your web app, you install the node-module **web3** via a command like “npm i -S web3” and it will save to your package.json file, if you are using one. So you would have, inside a “\” tag or inside a .js file, a line like this to load the web3 module itself:

```
import Web3 from 'web3';
```

(or, in ES6 javascript)

```
var Web3 = require('web3');
```

We have to tell Web3 about some details about our node and our smart-contract, so it can connect. You'll see lines like this:

```
# this line specifies our localhost node IP:port settings
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));

# our smart-contract address, from above
var GeektAddress = '0xe70ff0fa937a25d5dd4172318fa1593baba5a027';

# our smart-contract ABI, as raw javascript object, not quoted string!
var GeektABI = [{"constant":true,"inputs":[],"name":"getUsers","outputs":[{"name":""," (...)}

# this loads the smart-contract into our web3 object
GeektContract = web3.eth.contract(GeektABI).at(GeektAddress);

# this finds the default account listed for the node connection (account 0 from TestRPC above)
defaultAccount = web3.eth.accounts[0];
Here are some lines that demonstrate read-only functions interfacing with our smart-contract, via web3:

# the below function is activated when the page loads, to get the "white-pages" of all known user addresses
GeektContract.getUsers(function(err,usersResult){

# the below function grabs an image as needed, and stores them in browser memory...
GeektContract.getImage(imageHash,function(err,imageResult){
```



This simple demo ÐApp just loads all the known user addresses and grabs the actual user image details and the image records themselves as needed on-the-fly. Here's a screenshot:

At this point (image above) I've refreshed my TestRPC client and just done the "truffle migrate" command which deploys the smart-contract against my localhost TestRPC node. You can see the interface is ready for me to click the button and "register" the first user. When I do this button click, my registerNewUser() function from my smart-contract will be called, adding this user-data to my (localhost testing) Ethereum blockchain node at the smart-contract address we noted above. Adding user-data to the blockchain "storage" will cost gas, and we need to figure out how much gas we should pay, instead of just taking a scientific wild-ass guess (SWAG). Here's the code that actually gets called when this "Sign Guestbook" button gets clicked:

```

GeektContract.registerNewUser.estimateGas(
  outerThis.state.defaultHandle,
  outerThis.state.defaultCity,
  outerThis.state.defaultState,
  outerThis.state.defaultCountry,
  {from:defaultAccount},
  function(err, result){

    ...

    var myGasNum = result;
    GeektContract.registerNewUser.sendTransaction(
      outerThis.state.defaultHandle,
      outerThis.state.defaultCity,
      outerThis.state.defaultState,
      outerThis.state.defaultCountry,
      {from:defaultAccount, gas: myGasNum},
      function(err, result){

```

So you can see, when we do a state-changing transaction that we actually have to pay for with Ethereum gas (real or testnet gas), we will call this **registerNewUser.estimateGas()** function before-hand, and then use that estimated amount of gas when we call the state-changing function for real, using the **registerNewUser.sendTransaction()** function.

When we started TestRPC and it spits out some test wallet addresses for us, these accounts are akin to the account you would have if you ran a full node and did a `getAccounts` RPC command against it, like a full-node wallet. TestRPC gives you some free testnet coins, 100 testnet Ether per account when it starts up.

When we call these state-changing functions like **registerNewUser()** or **addImageToUser()**, in localhost testing phase, against our TestRPC node, TestRPC just pays the testnet gas for you and returns the result of the transaction right away. On a full localhost node, you would have to “unlock” that account before the transaction would succeed on the mainnet, and you would have to wait on the mainnet for the next block to pick up your transaction, assuming our gas estimate was enough! But there’s always a better way, my homies 😊

Hooking your Ðapp to the Mainnet

I’ve set up and deployed smart-contract onto the Ethereum mainnet, and this demo app on one of my web-servers, at www.enledger.io/ethereum-demo-tools/ (<https://www.enledger.io/ethereum-demo-tools/>). If you visit this site and you don’t have a TestRPC or full Ethereum node running on localhost:8545, and you visit this site, you’ll likely see something like this:

```

Saw connection to network: !
Saw default account: (no web3.eth node link)

```

So, does this mean you need a full Ethereum node running, or a Virtual Private Server (VPS) full node to connect to, in order to interact with the Ethereum Mainnet? Until just recently that was true, but now we have this great chrome plugin called Metamask (<https://metamask.io/>) (for Chrome browser), which allows you to connect to the Ethereum Mainnet within your browser, and the Metamask guys basically provide a connection to a full node for you, right there, for free, so props to them!

You can see in my code I detect for the special “injected” web3 object from the metamask plugin that sets up the node connection to their Ethereum full node. Here’s the relevant code that switches automatically to your regular client (localhost) or metamask web3 connection:

```

function loadWeb3() {
  let web3Injected = window.web3;
  if(typeof web3Injected !== 'undefined'){
    console.log("saw injected web3!");
    web3 = new Web3(web3Injected.currentProvider);
  } else {
    console.log("did not see web3 injected!");
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
  }
}

```

In the image below, note the Metamask Fox logo in the upper right corner, I’ve installed Metamask Plugin and connected to the Ethereum Mainnet in my browser! You can see I’ve added a couple users to the guestbook already in our “Geekt” Guestbook demo smart-contract. The ÐApp interface allows you to select users and browse their images, and you’ll see the Sign Guestbook page if you connect from an address that has not registered a user yet.

I added some code so you can just paste in the URL of an image you want to add to your user account, and it will fetch the SHA256 notary hash for you to add to the image, and it will also check the SHA256 notary hashes of images as they are loaded into the page, and give you a little green

checkmark or a red 'X' based on whether the image at that URL and the notary hash still match. I'm adding the "BlockGeeks" logo image under my Ryan M. user account here, and I've clicked the button for to add the image for this user.

The Metamask plugin has detected that I need to pay for this web3 Ethereum transaction with real Ethereum gas, and it has popped up a little window to prompt me whether I will accept this transaction. I've sent a little real Ether to this Metamask wallet, and it has enough to clear the transaction, so I click 'accept'.

Final Thoughts, Deep Reflections

That's it, it's working! This demo article and code repository should greatly help you along your to becoming a great Ethereum Solidity Language and DApp developer! You can install the Metamask Plugin, connect to Ethereum Mainnet, and visit www.enledger.io/ethereum-demo-tools (<https://www.enledger.io/ethereum-demo-tools/>) and you'll see my Guestbook / Notary Demo app and these same users and images, and you can even sign the guestbook and notarize images too, if you want 😊

Just so you guys don't get too out-of-control with this thing, I added a little moderation ability to the smart-contract "owner", which I have set internally to the address of whoever initially paid to deploy the smart-contract to the mainnet, which is me. I want to show you really quickly how that special admin ability works because it's pretty common to see this sort of admin user code pattern:

```

address GeektAdmin;
function Geekt() payable { // this is the CONSTRUCTOR (same name as contract) it gets called ONCE only when contract is
    GeektAdmin = msg.sender; // just set the admin, so they can remove bad users or images if needed, but nobody else can
}
modifier onlyAdmin() {
    if (msg.sender != GeektAdmin)
        throw;
    // Do not forget the "_;"! It will be replaced by the actual function body when the modifier is used.
    _;
}
function removeUser(address badUser) onlyAdmin returns (bool success) {
    delete Users[badUser];
    return true;
}
function removeImage(bytes32 badImage) onlyAdmin returns (bool success) {
    delete notarizedImages[badImage];
    return true;
}
}

```

We have a special constructor function, `Geekt()`, with the same name as the contract itself, that gets called only once, when the contract is first deployed to the blockchain. This function sets the admin user to the address of the `msg.sender` who paid to deploy the smart-contract. We also have a special `onlyAdmin()` modifier function that is then applied to our `removeUser()` and `removeImage()` functions, to restrict those functions so that they can only be activated if the `msg.sender` address is the admin user address.

Now I have a way to remove all the badUsers, if necessary 😊 Don't make me do it!

Another last point I want to mention is that we've only really scratched the surface here. One sort of basic feature I didn't cover is Solidity "events", which is a way for you to push updates back down to the browser, sort of like a "socket" connection watching and activating browser events when a new registered users or image is detected. The really coolest parts of Ethereum and Solidity come from smart-contracts that interact with Oracles, and smart-contracts that talk and act amongst each other using "messages" (DAO's). There's also some

The really coolest parts of Ethereum and Solidity come from smart-contracts that interact with Oracles, and smart-contracts that talk and act amongst each other using "messages" (DAO's). There's also some *pretty serious* security considerations you should learn about, before you go trying to store real ethereum within your smart-contract and send it out to people based on function calls, a bunch of people lost about \$60M dollars over that one and it ended up splitting the Ethereum network in a contentious hard-fork. That's something you should really watch out for, and I should probably cover that in another article, of course 😊 This article should be good to get you going and be "dangerous" as I said, and I encourage you to learn it, even get some nice company to pay you to learn it if you can, but there's still more to this path.

Another, the last point I want to talk about is the very structure of Ethereum and the real cost of using smart-contracts, right now. Ethereum is a network that runs on a big, single, public blockchain, where everyone can pay to access and store data in there. However, actually *doing* that is somewhat expensive. You can see above in my image, I was being charged 0.004182 Ether, which today is equivalent to \$0.96 in real US dollars, to store an image URL, SHA256 notary hash, and timestamp, comprising 196 bytes, to the Ethereum Mainnet. This works out to a data storage cost of \$5,159.03 per MB!

That's literally *insanely* expensive. The original idea for gas cost described in the Ethereum Whitepaper says that gas cost is ideally supposed to stay somewhat constant, however gas cost is tied to blocknumber, in the real implementation, and the block number is not going up nearly as fast as the current market price of Ethereum, so gas is getting way more expensive in real terms. Also, the hard-fork situation shows that this really is a public chain, and if something really contentious happens on it, it could fork and your data could theoretically be subject to rollback, or the underlying asset class could drop in price steeply. The sheer expense of data, and the sheer oceans of data out there waiting to be stored means that the amount of data storable in any chain might need to be limited, or may be self-limiting.

Ethereum might be best suited for things that need only a small amount of publicly-available data-storage to be on-chain, like maybe an online reputation system, or a data notary project. It may not make sense to build a blockchain project to put all the US widget-industry data onto the Ethereum blockchain, for example, because you might not want all that info publicly available, and you need to get those transaction fees way down for your widget-industry-specific usages. You might consider that a proof-of-stake blockchain model may be more energy-efficient, even if that may represent a theoretical weakening of the consensus security model versus Nakamoto proof-of-stake for your blockchain project.

I suggest that before embarking on a new blockchain project, consider carefully the platform you want to use, look at the newest stuff out there and don't be afraid to explore, and also consider carefully the *node incentive* that will attract people to download your cool node code, and run it! The best compliment a programmer can get is simply people using their code in the real-world, and being productive with it.

Secret Crazy Bonus Section

[Deploy me up to the Mainnet, Captain](#)

I'm back, with more awesome stuff for you and the Ethereum community! So I sort of skipped an important detail above, you may have noticed. Once you write your smart-contract, test compiling it a thousand times until it works, deployment, localhost test all works. Now, how the *hell* do you get this thing onto the Mainnet!? It's sort of

It's sort of difficult and usually involves having to run your own full Ethereum node, or pay someone to run one for you and allow you to access it (like via a VPS). So, that was really inconvenient for me, because I'm impatient and I am writing this article under a deadline, people! I don't have time to wait for my new Ethereum full node to sync, and I don't feel like blasting away a couple hundred gigs of hard-drive space and one of my boxes to crunching Ethereum tx full-time, and I'm also too cheap to pay for a VPS to do it for me, lol.

So I used some special javascript web kung-fu, and just figured out how I can make a web page with a box and a button, where you can just paste your contract, click the button, and deploy a Solidity smart-contract directly to the Mainnet, via Metamask Plugin. Mainly I did this just for the convenience, to write this article, but it turned out to be a surprisingly useful tool, so I put it up on the web for everyone to use. For the first time, you can do all your testing and even full smart-contract deployment to the Ethereum Mainnet, without needing a full Ethereum node of your own! Let me introduce the **EthDeployer** tool, it's freeware for you guys because free software gives us freedom 😊

Here's a link to the EthDeployer Repo on Github (<https://www.github.com/tectract/ethdeployer>), and you can also simply access my live running Tectract's EthDeployer (<http://enledger.io/EthDeployer>) and deploy your contracts directly from there, you just need to install Metamask Plugin and connect to the Ethereum Mainnet first, plus you'll need a little Ether in your Metamask wallet to pay the Mainnet smart-contract deployment fee too, of course!

This tool uses the Browser-Solc (<https://github.com/ericxtang/browser-solc>) tool, which is just a browserified, minified loader tool for loading up javascript version of Solidity Compiler, directly into the client-side browser, for parsing / compiling a smart-contract on-the-fly. This makes the tool completely stand-alone and portable, meaning you don't need to worry about installing the Solc compiler on your computer, and I don't need to worry about installing it on my server either, which is excellent for everyone. Let me point out the code that loads the latest available solidity compiler:

```
<script src="/browser-solc.min.js" type="text/javascript"></script>
```

I have the minified browser-solc.min.js javascript loading via the top-level index.html page, which makes a *window.BrowserSolc* object available to my lower-level react scripts. This is another very simple create-react-app that can be installed and deployed in minutes on your own machine, I even provided an actually useful readme.md in the github repo showing you how to do that.

```
setupCompiler(){
  var outerThis = this;
  setTimeout(function(){
    // console.debug(window.BrowserSolc);
    window.BrowserSolc.getVersions(function(soljsonSources, soljsonReleases) {
      var compilerVersion = soljsonReleases[_.keys(soljsonReleases)[0]];
      console.log("Browser-solc compiler version : " + compilerVersion);
      window.BrowserSolc.loadVersion(compilerVersion, function(c) {
        compiler = c;
        outerThis.setState({statusMessage:"ready!"},function(){
          console.log("Solc Version Loaded: " + compilerVersion);
        });
      });
    });
  },1000);
}
```

The setupCompiler function waits for a second for the window.BrowserSolc object to be available after the page loads, then does it's internal .getVersions() and .loadVersion() functions, there's really not much to it, and we have a functional Solc compiler available to us directly in a fully client-side environment, nice! For completeness, I'll show some relevant lines that actually handle compiling and deploying the contract from within a javascript function, when the "compile & deploy!" button is pressed:

```
compileAndDeploy() {
  ...
  var result = compiler.compile(this.state.contractText, optimize);
  var abi = JSON.parse(result.contracts[_.keys(result.contracts)[0]].interface);
  var bytecode = "0x" + result.contracts[_.keys(result.contracts)[0]].bytecode;
  var myContract = web3.eth.contract(abi);
  ...
  web3.eth.estimateGas({data: bytecode},function(err,gasEstimate){
    ...
    myContract.new({from:web3.eth.accounts[0],data:bytecode,gas:inflatedGasCost},function(err, newContract){
      ...
      thisTxHash: newContract.transactionHash,
      thisAddress: newContract.address
    })
  })
}
```

thisAddress: newContract.address

We have all our familiar objects from before, we call `compiler.compile()` on the contract text, and get a compiled contract “result” object from which we extract the abi and bytecode, and send it off in a new transaction. We see our old familiar `.estimateGas()` function thrown in here for good measure too. We grab the transaction ID and the new contract address to display back to the user when the contract is successfully deployed. Bam, done! Let’s see it in action:

“Oh yeah, if you get an error message on deployment, make sure to actually check the transaction ID link, the contract most likely DID actually deploy successfully, sometimes Metamask just fails to pick up the deployment success transaction state and new contract address and return those to our browser environment successfully.”

Saw connection to network: **mainnet!**

Saw default Eth account to use: [0x510d9bc3035666361909bd4d076c91926c207eb8!](#)

```
function getUsers() constant returns (address[]) { return usersByAddress; }  
function getUser(address userAddress) constant returns (string,bytes32,bytes32,bytes32,bytes32[]) {  
    return (Users[userAddress].handle,Users[userAddress].city,Users[userAddress].state,Users[userAddress].count  
}  
function getAllImages() constant returns (bytes32[]) { return imagesByNotaryHash; }  
function getUserImages(address userAddress) constant returns (bytes32[]) { return Users[userAddress].myImages;  
function getImage(bytes32 SHA256notaryHash) constant returns (string,uint) {  
    return (notarizedImages[SHA256notaryHash].imageUrl,notarizedImages[SHA256notaryHash].timeStamp);  
}  
}
```

Contract Deployed to mainnet

Compile & Deploy

new contract TXID: [0x949763bc3f64213a947a538c733e0c6a15361f74caaab408858e1030cfacb71a](#)
new contract address: [0x93606e40881b4d911b746fef659f35836726668b](#)

Account 1
0x510D9...
0.244347 ETH
BUY SEND

HISTORY

June 01 2017 00:25	Contract Published	0 ETH
May 31 2017 22:32	Contract Published (Rejected)	0 ETH
May 31 2017 21:41	0x6F283cA1...1c90	0 ETH
May 26 2017 04:21		

It's still alive. This makes me so happy! Secret Crazy Extra Bonus Unlocked! So, I paid 0.02268 Ether in gas, which is about \$5.30 USD (June 1st, 2017), to deploy this contract onto the Ethereum Mainnet. Remember this was supposed to be a super-simple demo app, with literally just two possible data structs. It was only like \$2 a couple weeks ago, this is wild! But still affordable, you shouldn't really need to deploy your contract to the main-chain that often, really only once should do it.

That's all I have for you this time, guys! Hope you found this tutorial super-useful, and that you can put good use to the EthDeployer tool in your dev adventures out there. I look forward to hearing questions and comments, thanks!

Ryan Molecke

Ryan Molecke, Chief Technology Officer (CTO) of EnLedger Corp. (<http://www.enledger.io>), is a highly multi-disciplinary academic computational science researcher, who transitioned into the world of finance technology and trading systems. He's worked at several notable and successful finance-tech startups, helped architect trading systems and brokerage/exchanges, and is now focused on permissioned ledgers and blockchain systems integrations, tokenizations, and credits programs. He holds a B.S.E. in Computer Engineering and a Ph.D. in Nanoscience and Microsystems Engineering, both from the UNM

Join over 115,115 Members

Angel Investors, Startups & Blockchain developers...

Enter your email address

Yes, Get Access!

No Thanks!



7



661




328



392

1K
SHARES

 Join the conversation



Dmitry Buterin @dmitry-buterin
(<https://blockgeeks.com/author/dmitry-buterin>) 6 months ago

What at awesome article!!!!

1

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)



Patricia Merkle @merklepatricia
(<https://blockgeeks.com/author/merklepatricia>) 6 months ago

Definitely, nice work, thank you for the article and for EthDeployer!

1

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)



jossiecalderon @jossiecalderon
(<https://blockgeeks.com/author/jossiecalderon>) 6 months ago

The struct ``User`` can be made more memory efficient, and no extra computing power is needed.

Instead of making the `myImages` type ``bytes32[]``, create it as type ``int[]`` where the location of the user's images in ``imagesByNotarizedHash`` gets mapped onto ``myImages``.

Now you just have to create a function that takes you from ``myImages`` to ``imagesByNotarizedHash``. Nothing has changed except you are now using ``myImages`` as a list of "pointers", and you are using a smaller data type to store them. I found this article most interesting in that it made me think about the code.

2

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

lowkeyenergy @lowkeyenergy
(<https://blockgeeks.com/author/lowkeyenergy>) 6 months ago

stfu nerd . you're missing the point .

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

jossiecalderon @jossiecalderon
(https://blockgeeks.com/author/jossiecalderon) 6 months ago

Oh, I'm sorry. What is the point?

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

Ryan Molecke @tectract
(https://blockgeeks.com/author/tectract) 5 months ago

myImages is a bytes32[] array so that the index is always the sha256 hash of the image being notarized.

This consideration is key here:

> We allow users to add their notaryHash to the global white-pages of images only if it doesn't exist, to prevent users from fighting over entries in the global white-pages, but we allow them to add/update any notaryHash entry within their own mini-whitepages of their own images.

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

leopoldjoy @leopoldjoy
(https://blockgeeks.com/author/leopoldjoy) 4 months ago

Thanks for the tutorial! Heads up, it should be "ethereumjs-testrpc" instead of "ethereum-testrpc"

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

Ryan Molecke @tectract
(https://blockgeeks.com/author/tectract) 3 months ago

As author of this article, I formally RESCIND permission for it to be published on this site. I'm using a CEASE AND DESIST statement to Ameer Rosic, asking him to remove this article from this site, due to personal attacks on my character.

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

jacobcolling @jacobcolling
(https://blockgeeks.com/author/jacobcolling) 2 months ago

Hello! Quick question, just about everything works but when I go to load the react app at localhost:3000 I get this error: "TypeError: web3.version.getNetwork is not a function"

Any idea what I should be doing differently?

The full output is below, thank you!

x

TypeError: web3.version.getNetwork is not a function

App.getInfo

src/App.js:148

145 | 13

```
145 | }
146 | }); */
147 |
> 148 | web3.version.getNetwork((err, netId) => {
149 |   var tempNetId = "
```

```
150 | if(err) {
151 |   tempNetId = err;
View compiled
(anonymous function)
src/App.js:802
799 | loadWeb3();
800 | setTimeout(function(){
801 |   // console.log("hello");
> 802 |   outerThis.getInfo();
803 | }, 1000);
804 | }
805 | }
View compiled
```

1

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

Ryan Molecke @tectract
(<https://blockgeeks.com/author/tectract>) 2 months ago

is it possible you may have forgotten to do 'npm i' in the app folder, or that the 'npm i' command didn't succeed on your machine?

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

umeboshi @umeboshi
(<https://blockgeeks.com/author/umeboshi>) 2 months ago

Great article, thank you.

I am stuck on getUser().

I choose the address from the Geekt deployment output here :

```
Deploying Geekt...
... 0x7884416783f2c0ddf8b6040be735e3f2f86811ce1fc3a8ea4874122306d1f7e2
Geekt: 0x9636ea0b3dc750995ac403284177824beeb37277
Saving successful migration to network...
```

And then I run the following

```
> Geekt.then(function(instance){return instance.registerNewUser("Tectract","Denver","CO","USA");})

> Geekt.then(function(instance){return
instance.addToImageToUser('www.myimageURL.com','0x6c3e007e281f6948b37c511a11e43c8026d2a16a8a45fed4e83379b66b0ab927');})

## What is hash here ? Is it hash of image in the url ? Can I specify bogus url for test purposes ?
```

```
truffle(development)> Geekt.then(function(instance){return instance.getUser('0x9636ea0b3dc750995ac403284177824beeb37277');})
[
  '0x0000000000000000000000000000000000000000000000000000000000000000',
  '0x0000000000000000000000000000000000000000000000000000000000000000',
  '0x0000000000000000000000000000000000000000000000000000000000000000',
  []
]
```

#Why might it not be registering ?

Thanks again for any help

I nanks again for any neip

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

kbayram @kbayram

(<https://blockgeeks.com/author/kbayram>) 2 months ago

thanks

0

Log in to Reply (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F)

You must be logged in (https://blockgeeks.com/wp-login.php?redirect_to=https%3A%2F%2Fblockgeeks.com%2Fguides%2Fsolidity%2F) to post a comment.

[Support \(https://blockgeeks.com/contact/\)](https://blockgeeks.com/contact/)

[Terms \(https://blockgeeks.com/guidelines/\)](https://blockgeeks.com/guidelines/)

[Privacy Policy \(https://blockgeeks.com/policy/\)](https://blockgeeks.com/policy/)



(<https://www.linkedin.com/company/blockgeeks/>) (<https://twitter.com/blockgeeks/>) (<https://www.instagram.com/blockgeeks/>) (<https://facebook.com/blockgeeks/>)

© Blockgeeks 2017

