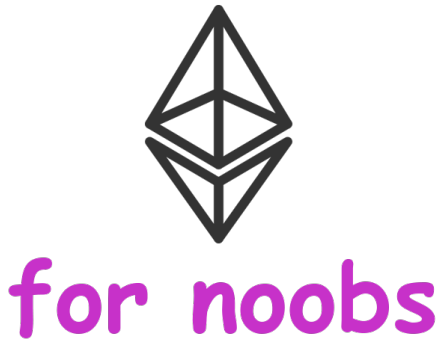
ConsenSys
A blockchain venture production studio building decentralized applications on Ethereum. Go to www.consensys.net and subscribe to our newsletter.
Oct 29, 2015 · 27 min read



A 101 Noob Intro to Programming Smart Contracts on Ethereum

> *Originally published at consensys.github.io/developers (where some of the code formatting might be easier to read).*

Some people say Ethereum is too logic-heavy and hard to use, but here's a write-up to give you a feel for building smart contracts and applications with it. Tools, wallets, applications and the ecosystem are still in development and it'll get easier!

- **Part I** is an overview of key terms and discusses **Ethereum Clients and Smart Contract Languages**.
- **Part II** discusses overall workflow and some current **DApp Frameworks and Tools** and
- **Part III** is the **Programming Part**, a quick walkthrough of writing tests and building a DApp for a smart contract using Truffle.

Part I. Intro

If you're new to *all* this cryptocurrency stuff, including Bitcoin and how it works, check out the first couple chapters of Andreas Antonopoulos' Bitcoin Book to dip your toe in the water. Then head over to the Ethereum Whitepaper.

If you start getting into some murky sections and would rather build something to get familiar first, then just read on. You don't have to understand all the crypto economic computer science to start building, and a lot of that paper is about Ethereum's improvements over Bitcoin's architecture.

Starter Tutorials

The official place to start is ethereum.org which has a starter tutorial and follow-up token and crowdsale tutorials. There's also the official Solidity docs. Another good place to start with smart contracts (where I started) is dappsForBeginners, although it might be outdated.

be outdated.

The goal of this write-up is to complement those tutorials and introduce some helpful dev tools that make starting out with Ethereum, smart contracts and building DApps (decentralized apps) easier. And to try to explain the overall flow of what's going on. This is from my (still-noob) perspective and with much help from the cool developers at ConsenSys.

Basic Concepts

It'd be good to know some of these terms:

Public Key Cryptography. Alice has a public key and private key. She can use her private key to create a digital signature, and Bob can use Alice's public key to verify that a signature is really from Alice's private key, i.e., really from Alice. When you create an Ethereum or Bitcoin wallet the long '0xdf…5f' address is a public key and the private key is stored somewhere. A Bitcoin wallet service like Coinbase stores your wallet's complementary private key for you, or you can store it yourself. If you lose your private key for a wallet with real funds you'll lose all your funds forever, so it's good to back up your keys. It hurts to learn this the hard way! I've done it.

Peer-to-Peer Networking. Like BitTorrent, all Ethereum nodes are peers in a distributed network, there's no centralized server. [In the future, there'll be hybrid semi-centralized services for Ethereum as a convenience to users and developers, more on that later.]

Blockchain. Like a global ledger or simple database of all transactions, the entire history of all transactions on the network.

Ethereum Virtual Machine. So you can write more powerful programs than on top of Bitcoin. It refers to the blockchain, what executes smart contracts, everything.

Node. Using this to mean you can run a node and through it read and write to the Ethereum blockchain, i.e., use the Ethereum Virtual Machine. A full node has to download the entire blockchain. Light nodes are possible but in the works.

Miner. A node on the network that mines, i.e., works to process blocks on the blockchain. You can see a partial list of live Ethereum miners here: stats.ethdev.com.

Proof of Work. Miners compete to do some math problem. The first one to solve the problem (the next block on the Blockchain) wins a reward: some ether. Every node then updates to that new block. Every miner wants to win the next new block so are incentivized to keep up to date and have the one true blockchain everybody else has, so the network always achieves consensus. [Note: Ethereum is planning to move to a Proof of Stake system without miners eventually, but that's beyond noob scope.]

Ether. Or ETH for short. It's a real digital currency you can buy and use! Here's a chart from one of several exchanges for it. At the time of writing, 1 ETH is worth about 65 cents in USD.

Gas. Running and storing things on Ethereum costs small amounts of ether. Keeps things efficient.

DApp. Decentralized App, what applications using smart contracts are called in the Ethereum community. The goal of a DApp is (well, should be) to have a nice UI to your smart contracts plus any extra niceties like IPFS (a neat way to store and serve stuff in a decentralized network, not made by Ethereum but a kindred spirit). While DApps can be run from a central server if that server can talk to an Ethereum node, they can also be run locally on top of any Ethereum node peer. [Take a minute: unlike normal webapps, DApps may not be served from a server. They may use the blockchain to submit transactions and retrieve data (important data!) rather than a central database. Instead of a typical user login system, users may be represented by a wallet addresses and keep any user data local. Many things can be architected differently from the current web.]

For another noob angle on some of the concepts above here's a good read: Just Enough Bitcoin for Ethereum.

Ethereum Clients, Smart Contract Languages

You don't have to run an Ethereum node to write and deploy smart contracts. See Browser-based IDEs and APIs below. But if you're learning, run an Ethereum node, it's good to get to know as a basic component and not hard to set up.

Clients for Running an Ethereum Node

Ethereum has several different client implementations (meaning ways to run a node to interact with the Ethereum network) including C++, Go, Python, Java, Haskell, etc. Why? Different strokes for different folks (like how the Haskell one is supposedly mathematically verifiable), and it improves the security and ecosystem of Ethereum to have so many. There's also a gui-based client in development, AlethZero.

At the time of writing, I've been using geth, the Go language one (go-ethereum) and on other days a tool called testrpc that uses the Python client, pyethereum. [Update: A new popular tool we use in lieu of testrpc now is ethersim that uses ethereumJS. EthereumJS is a JavaScript client that doesn't support real blockchain mining, so it's not a full client like the others listed above, but mining can be simulated for testing and development purposes.] The later examples will involve those tools.

[Sidebar: I've also tried the C++ one and still use its ethminer component for mining along with geth as the node, so different pieces can work together. Sidebar on Mining: Mining can be fun, sort of like having a houseplant you tend to, and another way to learn about the ecosystem… even if the price of ETH right now is not worth the local electricity costs of mining, that may change. Especially if everyone starts building cool DApps and Ethereum becomes more popular.]

Interactive Console. Once you have a node using one of the clients, you can sync with the blockchain, create wallets and send and receive real ether. One way to do that with geth is through the JavaScript console. Another way is via JSON RPC (remote procedure calls) using a command like cURL for getting stuff via URLs. However the goal of this article is to walk you through a DApp development scenario so let's just move on. But these tools are good to remember for debugging, configuring nodes and using a wallet via command line.

Running a node on a test network. If you install a client like geth and run it on the live

Running a node on a test network. If you install a client like geth and run it on the live network, it will take a while to download the entire blockchain and sync with the network. (You can check that it's synced by seeing that you have the latest block which is listed at the top of stats.ethdev.com and comparing that number to the block number output by your client node's logs.)

However to run smart contracts on the live network you'd have to cough up some real ether. Instead there are ways to run clients on a local testnet for for free. They won't download the full blockchain and will create a private instance of the Ethereum network with its own blockchain, so are faster to use for development.

testrpc. You can run a test network using geth, or another fast way of getting a testnet running is using testrpc. Testrpc will create a bunch of pre-funded accounts for you that will be listed when it starts up. It's is also super fast, so easier to develop and test with. You can start with testrpc, then when your contracts are in good shape, move to geth on a testnet, which can be started by specifying a networkid like: geth—networkid "12345". Here's the testrpc repo but I'll review everything you need to install again in the tutorial part later. [Update: The developer of testrpc is now focusing on ethersim as a replacement for testrpc and I'll update this tutorial eventually to use ethersim too. You can start using it now in if you'd like. Ethersim is based on ethereumJS and simulates mining for dev purposes and is very fast.]

Let's talk about programming languages next, then we can dive into actually coding stuff.

Programming Languages for Smart Contracts

Just use Solidity. To write smart contracts there are a few different languages: Solidity, which is like JavaScript and has .sol as a file extension, Serpent, Python-like with extension .se, and a 3rd, LLL, based on Lisp. Serpent was popular a while back but Solidity is the most popular right now and more robust, so just use Solidity. You prefer Python? Use Solidity.

solc Compiler. After writing a contract in Solidity, use solc to compile it. It's from the C++ libraries (different implementations complementing each other again) which can be installed here. [If you don't want to install solc you can also just use a browser-based compiler like the Solidity real-time compiler or Cosmo, but the programming part later on will assume you have solc installed.]

[Note: Ethereum's libraries are undergoing active development and sometimes things get out of sync with new versions. Make sure you have the latest dev version, or a stable version. Ask in one of the Ethereum Gitter's on Github or forums.ethereum.org what version people are using if things that used to work stop working.]

web3.js API. Once a Solidity contract is compiled with solc and sent to the network, you can call it using the Ethereum web3.js JavaScript API and build web apps that interact with contracts. (No need to install this yet, read up on DApp Frameworks below first.)

Those are the basic Ethereum tools for coding smart contracts and interacting with them to build DApps.

Part II: DApp Frameworks, Testing and Workflow

Part II. DApp Frameworks, Tools and Workflow

DApp-building Frameworks

You can do all these steps with just the tools mentioned above, but some helpful devs have created DApp frameworks to make development easier.

Truffle and Embark. The one that got me started is Truffle. (Before Truffle I watched a group of smart student interns last summer code stuff for a sleepless hackathon (albeit with terrific results) and shrank back in fear. Then Truffle came along and did a lot of the nitty gritty stuff for you, so you can start writing-compiling-deploying-testing-building DApps right away.) Another very similar framework for building and testing DApps is Embark. Between those two, I've only used Truffle, but there are very successful DApp devs in both camps. [Update: Some other good dapp-building frameworks are Dapple and Populus. Dapple also just got a dev grant to be improved.]

Meteor. Another stack a lot of DApp devs use include web3.js + Meteor which is a general webapp framework (The ethereum-meteor-wallet repo has a good starter example, and SilentCiero is building a lot of Meteor integrations with web3.js and DApp boilerplates). I've downloaded and run cool DApps that do things this way. There'll be some interesting discussion of all of these tools and best practices for building DApps at Ethereum's ĐΞVCON1 conference Nov. 9–13th (which will also be streamed or on YouTube).

APIs. BlockApps.net is creating a RESTful API for DApps based on a Haskell node they run as a centralized service to save you the trouble of running a local Ethereum node. This departs from the completely decentralized model of DApps but is useful when running an Ethereum node locally isn't realistic. For example if you want to serve your DApp to users who won't be running local nodes either and reach a wider audience with just a web browser or mobile device. BlockApps has a command line tool called bloc in the works that can be used after creating a developer account with them.

If users have to run a local Ethereum node to use DApps isn't that a dealbreaker? Like BlockApps there are a range of tools in development so this won't be. Metamask lets you run Ethereum stuff in a browser without a node, Ethereum's AlethZero or AlethOne are easier-to-use GUI clients being developed and a LightWallet ConsenSys is building are ways to make interacting with DApps more painless. Light (SPV) nodes and sharding are also in the works or planned. It's a P2P ecosystem but can involve hybrid architectures.

Smart Contract IDEs

IDEs.There's a Mix IDE for writing contracts put out by Ethereum. Haven't tried it but will soon.

Browser-based IDEs. The Solidity real-time compiler and Cosmo are both a fast way to get started compiling your smart contracts right away in a browser. You can even point your local node at these hosted instances by opening up a port (you should trust the site and not have your life savings in ether on your local node for that! See the Cosmo UI for instructions on how to do this with geth). But once your contract is working ok it's nice to use a framework for adding a UI and packaging it all up as a DApp, which is what Truffle does and will be explained in the programming part later.

Another powerful enterprise-y browser IDE is in the works by Ether.Camp. Their IDE comes with a sandbox test network with an auto-generated GUI for testing (instead of writing tests manually as shown in the tutorial later) as well as a sandbox transaction explorer at test.ether.camp. When you're ready to deploy your contract for semi-real, using their testnet can be a good way to confirm your smart contract's working as expected on a closer-to-real testbed. The same explorer for the live Ethereum network is at frontier.ether.camp and it shows details about every transaction ever. Ether.Camp's IDE is invite-only for eager guinea pigs at time of writing but will be launched soon.

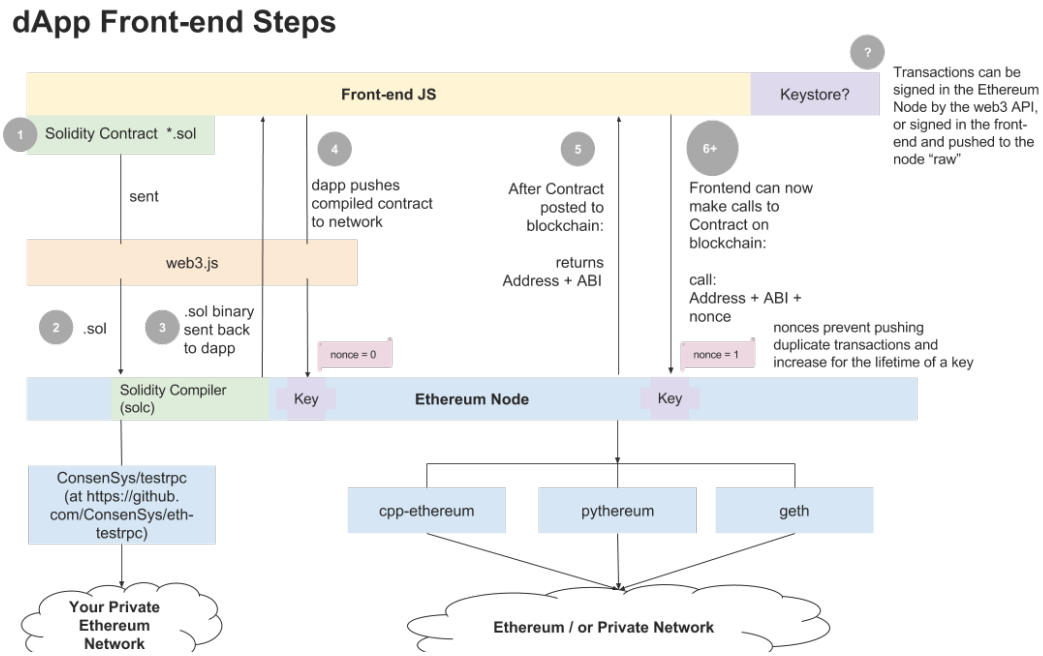
Sample Contracts and DApps. Search Github for DApp repos and .sol files to see what cool stuff people do and how. A big list of DApps with repos is also here: dapps.ethercasts.com, although some of the list's details are a little out of date. Ether.fund/contracts also has some examples of Solidity and Serpent contracts people have written, but not sure if these have been tested or verified for correctness. There'll be a whole day of DApp presentations Thursday, Nov. 12th at ĐΞVCON1.

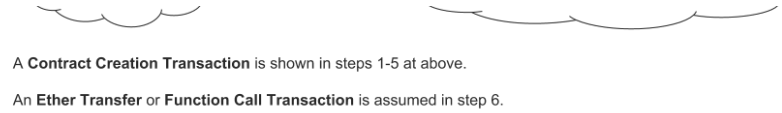
Workflow for Deploying Smart Contracts

The workflow is:

1. Start an **Ethereum node** (e.g. geth or testrpc or ethersim)
2. **Compile** your Solidity smart contract using solc => get back the binary
3. **Deploy** your compiled contract to the network. (This step costs ether and signs the contract using your node's default wallet address, or you can specify another address.) => get back the contract's blockchain address and ABI (a JSON-ified representation of your compiled contract's variables, events and methods that you can call)
4. **Call** stuff in the contract using web3.js's JavaScript API to interact with it (This step may cost ether depending on the type of invocation.)

This workflow is depicted in greater detail in the diagram below:





A **Contract Creation Transaction** is shown in steps 1-5 at above.

An **Ether Transfer** or **Function Call Transaction** is assumed in step 6.

You could build a DApp that provides a UI for users to deploy a contract then use it (Steps 1 or 4). Or your DApp could assume the contract's already been deployed (common) and start the UI flow from there (Step 6).

Part III. The Programming Part, Finally

Testing in Truffle

Truffle is great for test-driven development of smart contracts which is highly recommended to maintain sanity when you're starting to learn how things work. It's also useful as a way to learn to write promises in JavaScript, i.e., deferred and asynchronous callbacks. Promises are like "do this, then when that comes back, do that, and when that comes back, do this other thing…and don't keep us waiting while all that's going on, ok?" Truffle uses a JS promises framework called Pudding on top of web3.js (so it installs web3.js for you too).

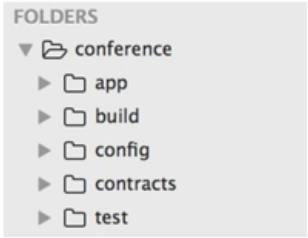
Transaction times. Promises are extremely useful for DApps because transactions need to be mined into the blockchain (takes 12–15 seconds in Ethereum). Even if they don't seem to take that long on a test network it may take longer on the live network, or to find out it didn't happen (e.g. your transaction could have ran out of gas, or was mined into a block that's been orphaned).

So let's copy a simple smart contract and write a test for it.

Using Truffle

Make sure you have 1. solc installed and 2. testrpc. (For testrpc you'll need Python and pip. If you're new to Python, to install it you may also need to use a virtualenv, a way to keep python libraries separate on a single computer.)

Install 3. Truffle (You can do this using NodeJS's npm: npm install -g truffle, the -g may require sudo). To verify it installed, type truffle list in a console window to list all truffle commands. Then create a new project directory (I'm naming my new directory 'conference'), change into it, and do truffle init. This will create this directory structure:



Now start a client node in **a new console window** by running testrpc (or start your geth node):

```
(venv)Evas-MacBook-Pro:testrpc Ohmu$ testrpc
No handlers could be found for logger "eth.pow"
Consensys TestRPC/0.1.18/python
Resetting EVM state...
```

Back in the first truffle console window, now type truffle deploy. This will deploy the Example contract truffle init created as boilerplate. Any errors messages you may have will show up in either the testrpc console window or the truffle window.

As you're developing you can do truffle compile to make sure your contracts compile (using solc you can also run solc YourContract.sol), truffle deploy to compile and deploy it to the network, and truffle test to run your smart contract tests.

First Contract, First Test

Here's a Solidity contract for a Conference where registrants can buy tickets, and the organizer can set a maximum quota of attendees as well as provide refunds. All the code presented in this tutorial is in this repo.

```
contract Conference {
  address public organizer;
  mapping (address => uint) public registrantsPaid;
  uint public numRegistrants;
  uint public quota;

  // so you can log these events
  event Deposit(address _from, uint _amount);
  event Refund(address _to, uint _amount);

  function Conference() { // Constructor
    organizer = msg.sender;
    quota = 500;
    numRegistrants = 0;
  }
  function buyTicket() public returns (bool success) {
    if (numRegistrants >= quota) { return false; } // see footnote
     registrantsPaid[msg.sender] = msg.value;
     numRegistrants++;
     Deposit(msg.sender, msg.value);
     return true;
  }
  function changeQuota(uint newquota) public {
    if (msg.sender != organizer) { return; }
    quota = newquota;
  }
  function refundTicket(address recipient, uint amount) public {
    if (msg.sender != organizer) { return; }
    if (registrantsPaid[recipient] == amount) {
      address myAddress = this;
      if (myAddress.balance >= amount) {
        recipient.send(amount);
        registrantsPaid[recipient] = 0;
        numRegistrants--;
        Refund(recipient, amount);
      }
    }
```

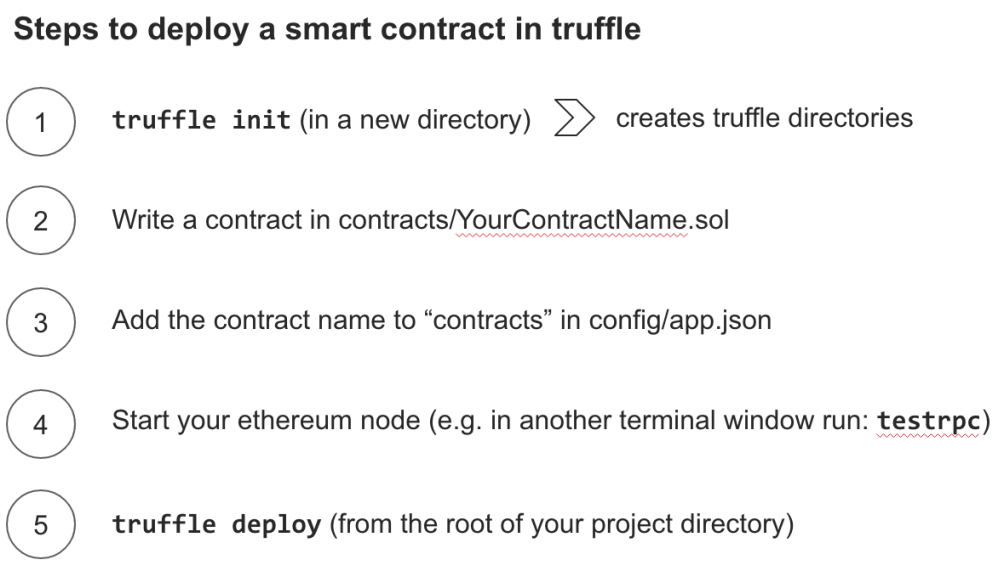
```
  }
  function destroy() { // so funds not locked in contract forever
    if (msg.sender == organizer) {
      suicide(organizer); // send funds to organizer
    }
  }
}
```

[Footnote: Currently if the quota is reached, buyTicket() keeps the money in the contract but the buyer won't get a ticket. So buyTicket() should use 'throw' to revert ticket buyer's transaction. I'll update the code and add a test for this soon. (Thanks to F.V. from the comments for catching this.)]

Let's deploy this.

[Note: At time of writing I have solc 0.1.3+ (installed via brew), Truffle v.0.2.3, testrpc v.0.1.18 (using a venv) on Mac OS X 10.10.5.]

Deploying the Contract



Add a new smart contract. After you've done truffle init, or in the existing project directory, copy-paste the Conference contract into contracts/Conference.sol. Then in the file config/app.json, edit the "deploy" array to include "Conference".

```
19 ▼  "deploy": [
20      // Names of contracts that should be deployed to the network.
21      "Conference"
22    ],
```

Start testrpc. Start testrpc in a separate console window using testrpc if its not already running.

Compile or Deploy. Run truffle compile to see if the contract compiles, or just do truffle deploy to compile and deploy at once. This will add the deployed contract's address and ABI(that JSON-ified version of the compiled contract) to the config directory, which

truffle test and truffle build will pull in from later on.

Errors? Did that compile? Again, error messages may show up in either the testrpc console or the truffle console.

Redeploy after restarting a node! If you stop your testrpc node, remember to redeploy any contracts using truffle deploy before trying to use them again. Each time testrpc restarts it's a blank slate.

Analyzing the Contract

Let's start with the variables at the top of the smart contract:

```
address public organizer;
mapping (address => uint) public registrantsPaid;
uint public numRegistrants;
uint public quota;
```

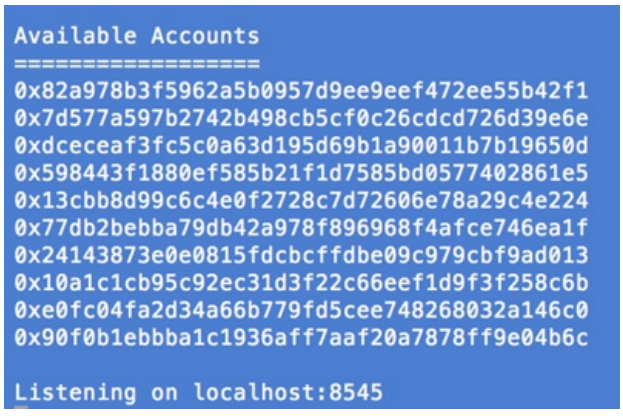
address. The first variable is the wallet address of the organizer. This is set when the constructor is called in function Conference(). A lot of contracts will also call this the 'owner'.

uint. An unsigned integer. Space is important on the blockchain so keep things as small as possible.

public. Means it can be called from outside the contract. A private modifier would mean it can only be called from within the contract (or by derived contracts). If you're trying to call a variable from a web3.js call in a test make sure its public.

Mappings or Arrays. Before Solidity added support for arrays, mappings like mapping (address => uint) were used. This could also be written as address registrantsPaid[] but mappings have a smaller footprint. This mapping will be used to store how much each registrant (represented by their wallet address) has paid so they can get refunds later on.

More on addresses. Your client node (i.e., testrpc or geth in these examples) can have one or more accounts. In testrpc, on startup an array of 10 "Available Addresses" are displayed:



The first one, accounts[0], is used by default for calling transactions if a different one is

not specified.

Organizer address vs. Contract address. Your deployed contract will have its own contract address (different from the organizer's address) on the blockchain. This address is accessible in a Solidity contract using this, as used inside the refundTicket function in the contract: address myAddress = this;

Suicide, a good thing in Solidity. Funds sent to the contract are held in the contract itself. In the destroy function above funds are finally released to the organizer set in the constructor. suicide(organizer); does this. Without it, funds can end up locked in the contract forever (somebody on reddit lost some ether this way), so make sure to include that method if your contract collects funds!

If you want to simulate another user or counterparty (e.g. simulate a buyer if you're a seller), you can use another address from the accounts array. To buy a ticket as a different user, say accounts[1], use it in the from field:

```
conference.buyTicket({
  from: accounts[1], value: some_ticket_price_integer });
```

Some Function Calls can be Transactions. Function calls that change the state of the contract (modify values, add records, etc.) are **transactions** and have implicit sender and value. So inside curly braces { from: __, value: __ } can be specified in a web3.js function call to send funds to a transaction function from a wallet address. On the Solidity end, you can retrieve these values using msg.sender and msg.value, which are implicity in Solidty transaction functions:

```
function buyTicket() public {
  ...
  registrantsPaid[msg.sender] = msg.value;
  ...
}
```

Events. These are totally optional. Deposit and Send in the contract are events that can be logged in the Ethereum Virtual Machine logs. They don't actually do anything, but are good practice for keeping track that a transaction has happened.

Okay, let's write a test for this smart contract to make sure it works.

Writing a Test

In your project folder's test/ directory rename the example.js test file to conference.js. Modify all instances of "Example" to "Conference".

```
contract('Conference', function(accounts) {
  it("should assert true", function(done) {
    var conference = Conference.at(Conference.deployed_address);
    assert.isTrue(true);
    done(); // stops tests at this point
  });
});
```

```
});
```

On running truffle test from the project's root directory you should see the test pass. In the test above truffle gets the contract's address on the blockchain from Conference.deployed_address.

Let's write a test to initialize a new Conference and check that the initial variables are being set correctly. Replace the test in conference.js with this one:

```
contract('Conference', function(accounts) {
  it("Initial conference settings should match", function(done) {
    var conference = Conference.at(Conference.deployed_address);
    // same as previous example up to here
    Conference.new({ from: accounts[0] }).then(
      function(conference) {
        conference.quota.call().then(
          function(quota) {
            assert.equal(quota, 500, "Quota doesn't match!");
          }).then(function() {
            return conference.numRegistrants.call();
          }).then( function(num) {
            assert.equal(num, 0, "Registrants should be zero!");
            return conference.organizer.call();
          }).then( function(organizer) {
            assert.equal(organizer, accounts[0], "Owner doesn't match!");
            done(); // to stop these tests earlier, move this up
          }).catch(done);
    }).catch(done);
  });
});
```

Constructor. Conference.new({ from: accounts[0] }) instantiates a new Conference by calling the contract's constructor. Since accounts[0] is used by default if no from is specified, so it could have been left out when calling the constructor:

```
Conference.new({ from: accounts[0] }); // same as Conference.new();
```

Promises. That's what those then and return's above are. What's going on above might start to look like a deeply nested function call chain like:

```
conference.numRegistrants.call().then(
  function(num) {
    assert.equal(num, 0, "Registrants should be zero!");
    conference.organizer.call().then(
      function(organizer) {
        assert.equal(organizer, accounts[0], "Doesn't match!");
          }).then(
            function(...))
              }).then(
                function(...))
                 // Because this would soon get hairy...
```

Promises flatten this to minimize nesting, allow for calls to return asynchronously and help simplify the syntax of writing "on success do this" vs. "on failure do that". Web3.js

provides callbacks for asynchronous requests (docs) so you don't have to wait for transactions to complete to do stuff in the front-end. (Truffle uses a promises framework wrapper to web3.js called Pudding, based on the framework Bluebird, which also has advanced promise features.)

call. Use this to check the values of variables as in conference.quota.call() or with an argument like call(0) to call a mapping and get index 0. Solidity docs say this is a "message call" which is 1. not mined and so 2. doesn't have to be from an account/wallet (therefore it's not signed with an account holder's private keys). Transactions on the other hand, are mined, have to be from an account (i.e., signed), and are recorded on the blockchain. Modifying any value in a contract is a transaction. Just checking a variable value is not. So don't forget to add call() when calling variables! Crazy things can happen. [Also, if you're trying to call a variable and having problems make sure its public.] call() can also be used to call functions that are not transactions. If they are meant to be transactions and you try to call() them, they won't execute as transactions on the blockchain.

assert. Standard JS testing assertion (if you type 'asserts' plural by accident truffle will have errors and you won't know what's going on), see the Chai docs for other types of assertions but assert.equal is usually all you need.

Run truffle test again to make sure that works for you.

Writing a Test calling a Contract Function

Let's test that the function that changes the quota works. Inside the contract('Conference', function(accounts) {…}; body of tests/conference.js stick this additional test:

```
it("Should update quota", function(done) {
  var c = Conference.at(Conference.deployed_address);
  Conference.new({from: accounts[0] }).then(
    function(conference) {
    conference.quota.call().then(
      function(quota) {
        assert.equal(quota, 500, "Quota doesn't match!");
      }).then(
        function() {
          return conference.changeQuota(300);
      }).then(
        function(result) {
          console.log(result);
          // printed will be a long hex, the transaction hash
          return conference.quota.call();
      }).then(
        function(quota) {
          assert.equal(quota, 300, "New quota is not correct!");
          done();
      }).catch(done);
  }).catch(done);
});
```

The new thing is the line that calls the changeQuota function. The console.log is also useful for debugging to print the result in the truffle console's output. Add a console.log to see if the execution gets to a certain point. Also make sure the changeQuota function in the Solidity contract is public, or you won't be able to call it:

```
function changeQuota(uint newquota) public { }
```

Writing a Test for a Transaction

Let's call a function that expects funds from a wallet address.

Wei. Ether has a lot of denominations (here's a helpful converter) and the one normally used in contracts is Wei, the smallest. Web3.js provides convenience methods for converting ether to/from Wei, as in web3.toWei(.05, 'ether'). JavaScript has issues with very big numbers so web3.js uses a BigNumber library and they recommend keeping things in Wei in your code until users see it (docs).

Account Balances. Web3.js provides a lot more convenience methods here, and another one used in the test below is web3.eth.getBalance(some_address). Remember that funds sent to the contract are in the contract itself until suicide is called.

Inside the contract(Conference, function(accounts) {…}; body stick this additional test. In the highlighted method below, the test has a second user (accounts[1]) buy a ticket for ticketPrice. Then it checks that the contract's balance has increased by the ticketPrice sent and that the second user has been added to the list of registrants.

In this test buyTicket is a Transaction:

```
it("Should let you buy a ticket", function(done) {
  var c = Conference.at(Conference.deployed_address);
  Conference.new({ from: accounts[0] }).then(
    function(conference) {
    var ticketPrice = web3.toWei(.05, 'ether');
    var initialBalance = web3.eth.getBalance(conference.address).toNumber();
    conference.buyTicket({ from: accounts[1], value: ticketPrice })
      .then(
        function() {
          var newBalance = web3.eth.getBalance(conference.address).toNumber();
          var difference = newBalance - initialBalance;
          assert.equal(difference, ticketPrice, "Difference should be what was sent");
          return conference.numRegistrants.call();
      }).then(
       function(num) {
          assert.equal(num, 1, "there should be 1 registrant");
          return conference.registrantsPaid.call(accounts[1]);
      }).then(function(amount) {
          assert.equal(amount.toNumber(), ticketPrice, "Sender's paid but is not listed");
          done();
      }).catch(done);
    }).catch(done);
});
```

Transactions are Signed. Unlike previous function calls this is a transaction sent funds, so under the hood the second user (accounts[1]) is signing the transaction call buyTicket() with their key. (In geth the user would have to enter a password to approve this transaction or unlock their account before sending funds.)

toNumber(). Sometimes results from Solidity returned have to be converted from hex. If it might be a really big number go with web3.toBigNumber(numberOrHexString)

it might be a really big number go with web3.toBigNumber(numberOrHexString) because Javascript can mess up big numbers.

Writing a Test for a Contract Sending a Transaction

Finally, for sanity, let's make sure the refundTicket method works and can only be activated by the organizer. Here's a test for it:

```
it("Should issue a refund by owner only", function(done) {
  var c = Conference.at(Conference.deployed_address);
  Conference.new({ from: accounts[0] }).then(
    function(conference) {
      var ticketPrice = web3.toWei(.05, 'ether');
      var initialBalance = web3.eth.getBalance(conference.address).toNumber();
      conference.buyTicket({
        from: accounts[1], value: ticketPrice }).then(
       function() {
        var newBalance = web3.eth.getBalance(conference.address).toNumber();
        var difference = newBalance - initialBalance;
        assert.equal(difference, ticketPrice, "Difference should be what was sent");
        // Now try to issue refund as second user - should fail
        return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[1]});
       }).then(function() {
       var balance = web3.eth.getBalance(conference.address).toNumber();
       assert.equal(web3.toBigNumber(balance), ticketPrice, "Balance should be unchanged");
       // Now try to issue refund as organizer/owner - should work
       return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[0]});
     }).then( function() {
       var postRefundBalance = web3.eth.getBalance(conference.address).toNumber();
       assert.equal(postRefundBalance, initialBalance, "Balance should be initial balance");
       done();
     }).catch(done);
  }).catch(done);
});
```

The corresponding Solidity function for the test above is here:

```
function refundTicket(address recipient, uint amount) public returns (bool success) {
  if (msg.sender != organizer) { return false; }
  if (registrantsPaid[recipient] == amount) {
    address myAddress = this;
    if (myAddress.balance >= amount) {
      recipient.send(amount);
      Refund(recipient, amount);
      registrantsPaid[recipient] = 0;
      numRegistrants--;
      return true;
    }
  }
  return false;
}
```

Sending ether from a contract. The address myAddress = this; line shows how to get the conference instance's address, so you can check the balance in the subsequent line (or just use this.balance). Also the recipient.send(amount) method is where the contract sends funds back to recipient.

Transactions cannot return results to web3.js. Note this! The refundTicket function returns a bool but this cannot be checked in your test. This method is a **transaction** (i.e., something that modifies values or send ether), and the result of a transaction to

web3.js is a transaction hash (if you printed the result it'll be a long hex/weird-looking object). So why add a return value to the refundTicket call at all? The return value can be read in Solidity, such as by another contract that calls refundTicket(). So Solidity contracts can read return values from a transaction, but web3.js transaction calls cannot. On the other hand, other contracts can't use Events (discussed below) which is how you can monitor transactions in web3.js, or check whether a transaction has modified instance variables in a subsequent test promise using call().

More on sendTransaction(). When you call a transaction like buyTicket() or refundTicket() using web3.js (which uses web3.eth.sendTransaction), the transaction does not execute right away. Instead the transaction is submitted to the network of miners, and the code does not run until one of those miners scores a block and the transaction is mined into the blockchain. So to verify a transaction you have to wait for it to make it onto the blockchain and then back to your local node. With testrpc this may be seem instantaneous because it's so fast but on a live network it will be slower.

Events. Instead of using return values you can listen for events in web3.js. The smart contract example has these events:

```
event Deposit(address _from, uint _amount);
event Refund(address _to, uint _amount);
```

And they are triggered in buyTicket() and refundTicket(). You can see these logged in the output of testrpc when called. To listen for them, you can also add web3.js listeners. At time of writing I haven't been able to log events inside of truffle tests but have logged them in an app:

```
Conference.new({ from: accounts[0] }).then(
  function(conference) {
    var event = conference.allEvents().watch({}, '');
    // or use conference.Deposit() or .Refund()
    event.watch(function (error, result) {
      if (error) {
        console.log("Error: " + error);
      } else {
        console.log("Event: " + result.event);
    }
});
```

Filters. Instead of checking all events above which may lead to a lot of polling, filters could be used instead. They allow you to start and stop watching for them when your transactions are done. More on filters can be found in the Solidity docs.

Overall, using events and filters are cheaper in terms of gas than checking variables so might be useful if you need to verify transactions on the live network.

Gas. Up to this point we haven't needed to discuss gas at all, it usually doesn't need to be explicitly set with testrpc. As you move to geth and then the live network it will. In your transaction calls you can send gas implicitly inside the {from: __, value: __, gas: __} objects. Web3.js has a call for checking the gas price web3.eth.gasPrice (The result

of this is not how much gas you should send with your transaction but the price of 1 unit of gas or step. For how much gas you should send for now its probably best to send something close to the gasLimit so its sure to run. Right now the maximum gas allowed in a block is 3141592. I just use 3000000). The Solidity compiler also has a flag you can call from the command line to get a summary of gas expenditures for your contract: solc —gas YourContract.sol. Here's the output for Conference.sol:

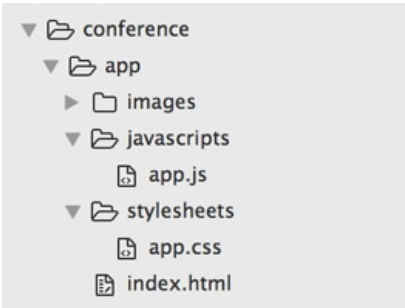
```
(venv)Evas-MacBook-Pro:contracts Ohmu$ solc --gas Conference.sol

======= Conference =======
Gas estimation:
construction:
   45426 + 258600 = 304026
external:
   registrantsPaid(address):    309
   organizer(): 277
   refundTicket(address,uint256):       [???]
   destroy():   379
   changeQuota(uint256):        20356
   quota():     328
   numRegistrants():    350
   buyTicket(): 41944
internal:
```

Creating a DApp UI for your contract

This next section assumes you might be new to some web development practices, just in case.

All the truffle tests written above are using JavaScript methods re-usable in a front-end UI. Add your UI to the truffle directory app/. On running truffle build it will be compiled along with contract configuration stuff to the build/ directory. When developing use truffle watch to constantly compile any changes in app/* to build/*. Then reload what's in the build/ directory in your browser. (truffle serve can also run a webserver for you from build/.)

In the app/ directory there'll be some boilerplate started for you:

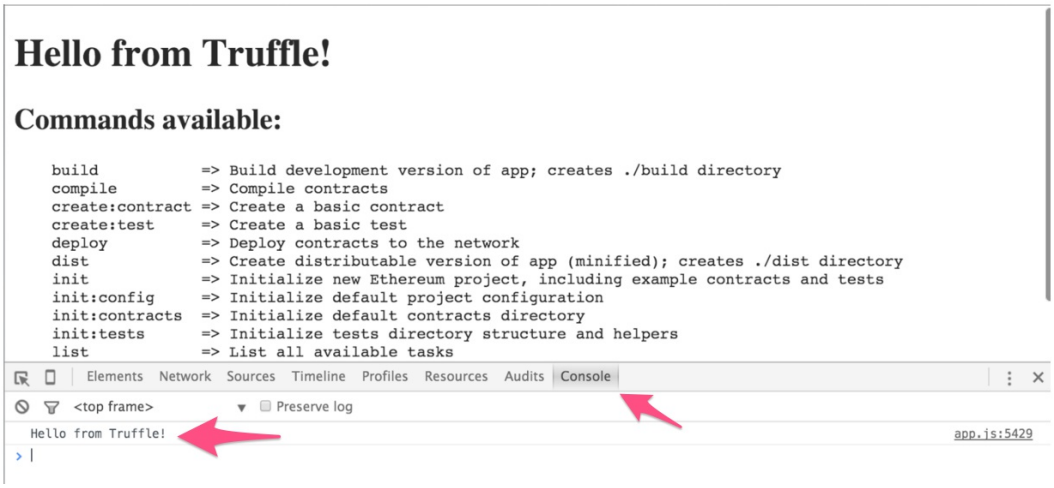


index.html already loads app.js:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Default Truffle App</title>
5    <link href="./app.css" rel='stylesheet' type='text/css'>
6    <script src="./app.js"></script>
7  </head>
```

So we can just add some code to app.js.

app.js has a console.log with “Hello from Truffle!” that will show up in your browser’s developer console. Start truffle watch in the project root directory and then open build/index.html in a browser window, and open the browser’s developer console (In a lot of browsers like Chrome, right-click » Inspect Element and switch to the Console tab below.)



In **app.js**, add a window.onload function that’ll be called when the page loads. This snippet below will confirm web3.js is loaded and show all the accounts available. [Note: your testrpc node should still be running.]

```
window.onload = function() {
  var accounts = web3.eth.accounts;
  console.log(accounts);
}
```

See if that prints an array of accounts to your browser console.

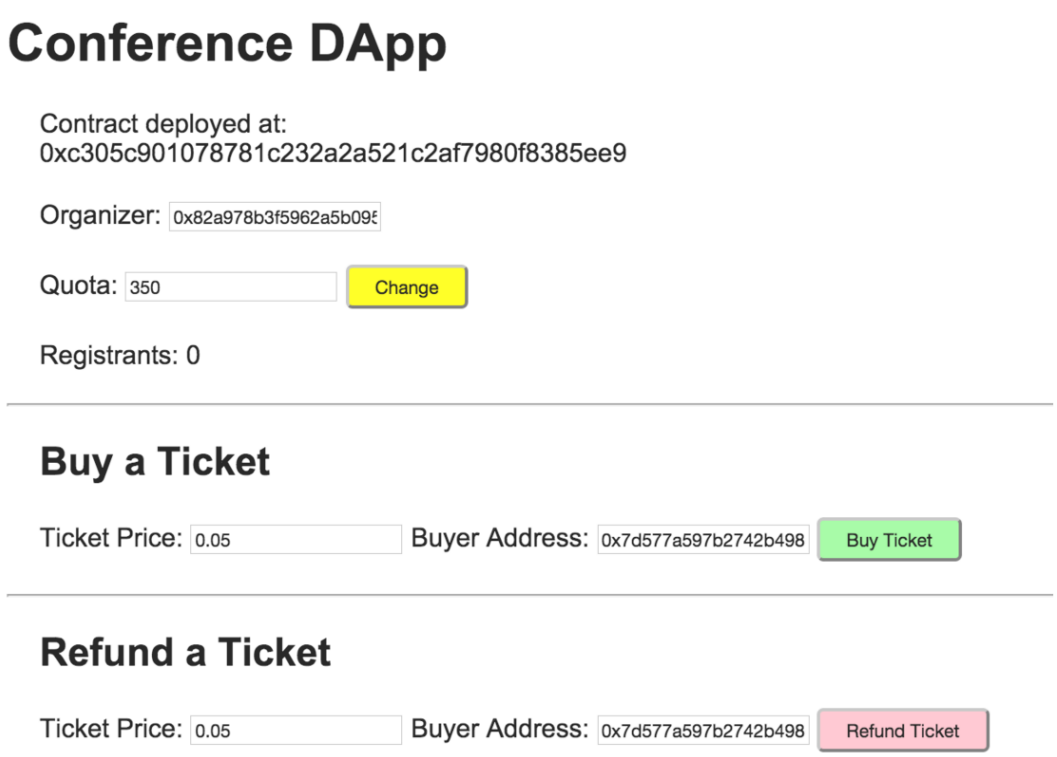
Now you can just copy some functions from tests/conference.js (remove the assertions since those are for testing), and output what’s returned to the console to confirm its working. Here’s an example:

```
window.onload = function() {
  var accounts = web3.eth.accounts;
  var c = Conference.at(Conference.deployed_address);
  Conference.new({ from: accounts[0] }).then(
    function(conference) {
      var ticketPrice = web3.toWei(.05, 'ether');
      var initialBalance = web3.eth.getBalance(conference.address).toNumber();
      console.log("The conference's initial balance is: " + initialBalance);
      conference.buyTicket({ from: accounts[1], value: ticketPrice }).then(
        function() {
          var newBalance = web3.eth.getBalance(conference.address).toNumber();
          console.log("After someone bought a ticket it's: " + newBalance);
          return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[0]});}).then(
        function() {
          var balance = web3.eth.getBalance(conference.address).toNumber();
          console.log("After a refund it's: " + balance); }); }); };
```

The code above should output:

```
The conference's initial balance is: 0
After someone bought a ticket it's: 50000000000000000
After a refund it's: 0
```

Now using whatever web tools you prefer, jQuery, ReactJS, Meteor, Ember, AngularJS, etc., you can start building a DApp UI in app/for interacting with an Ethereum smart contract! Below is a super simple jQuery-based UI as an example.



Here's the index.html. And here's the app.js.

Now that I'm interacting with the smart contract in a UI I'm realizing it'd be good to add checks to make sure the same user can't register twice. Also since this is running on testrpc, which is really fast, it'd be good to switch to geth and make sure the transactions are still responsive to the user. Otherwise the UI should have some loading messages and disabled buttons while the transactions are being processed if they're going to take a while.

Trying geth. If you're using geth, this line was working for me (geth v1.2.3):

```
geth --rpc --rpcaddr="0.0.0.0" --rpccorsdomain="*" --mine --unlock='0 1' --verbosity=5 --maxpeers=0 --minerthreads='4' --networkid '12345' --genesis test-genesis.json
```

This unlocks two accounts, 0 and 1. Note:
1. You may need to enter the passwords to both accounts after the geth console starts up.
2. You will also need a test-genesis.json file with both of your accounts well-funded

under ‘alloc’ in the test-genesis.json.)
3. Finally, for geth you may need to add gas when calling the constructor:

```
Conference.new({from: accounts[0], gas: 3141592})
```

Then re-do the whole truffle deploy, truffle build thing.

Code for this tutorial. Again, all the code presented in this tutorial is in this repo.

Auto-generating UIs from contracts. SilentCicero has also built a tool called DApp Builder to auto-generate from a Solidity contract HTML, jQuery and web3.js calls you can modify. This is also starting to be a common theme of some other smart contract dev tools coming out.

Ok tutorial over! This last part was a walkthrough of just one set of tools, mainly Truffle and testrpc. Even within ConsenSys, different developers use different tools and frameworks. You might find tools out there that are a better fit for you, and some things may change in a few months (Souptacular keeps an updated gitbook of resources and notes). But this workflow has helped me get started learning to build DApps.

(⊙ω⊙) wonk wonk

Thanks to Joseph Chow for a lot of proofreading and suggestions, as well as Christian Lundkvist, Daniel Novy, Jim Berry, Peter Borah and Tim Coulter for corrections and debugging help, and Tim Coulter, Nchinda Nchinda and Mike Goldin for helping with the DApp Front-end Steps diagram.

by Eva Shon
UX Designer
ConsenSys

Bitcoin Ethereum Blockchain