Udemy Ethereum and Solidty Course

Section 1

MetaMask

Main network

Test networks: Rinkeby (used in this course), Ropsten, Kovan

# Account Name: metamask name for accounts

# Account Address: Similar to an email address

# Public Key:

# Private Key: required to send from account/public key

Hexadecimal numbers…. Javascript converts to a base10 number for readability

Account address connect to all ethereal networks

Rinkeby-faucet.com to get test ether

Web3js on backend server -> ethereum blockchain -> server waits for transaction confirms, then send message back

Transaction:

Record of one account sending property rights to another account

Each transaction has a list of properties [

Nonce (nonce = none sense) number of how many times a sender has sent a transaction

To = address of the account property rights are going to

Value = Value of ETH

gasPrice = amount of ETH sender is willing to pay per unit of gas

startGas/gasLimit = units of gas that transaction can consume

V/R/S = sender’s key private key generates cryptographic pieces

External Accounts = users’ accounts that are cross network

Contract Accounts are specific to a network (deploy into Rinkeby test network, then if success deploy to main)

ETH Contract Account

Field [ description ]

Balance [ amount of ether this account owns ]

Storage [ data storage for contract ], a cubby where we can store some data for contract (numbers, strings, arrays)

Code [ raw machine code for contract ], machine level code (we write code that compiles to this level)

Smart Contract as a CLASS

With many instances on the blockchain

**Solidity**

Written in .sol files

Strongly typed (javascript is dynamic… )

Similar to javascript ish… but eh… not really

Has several huge, gigantic ‘gotchas’

Contract definition (solidity) we write -> solidity compiler [ byte code ready for deployment, application binary interface (ABI)]

The ABI is how the front end talks with the contract

Javascript talks to ABI, which then sends signals to the blockchain in bytecode

**First Contract**

Using ReMix IDE

Contract class named as inbox

define storage variable / instance variable that is stored on the blockchain (as opposed to local variable)

LOCAL variables are created to execute a contract, then thrown out at end of computation and never persists on blockchain

Constructor function is invoked automatically when deployed to blockchain

Functions for this contract:

Constructor functions = contracts contract when deployed to contract

— NOTE: use constructor instead of function. Functions as constructors have been depreciated

Regular functions:

| Func name | func type | returns type |

function functionName() public view returns (string) {

return variable;

}

Common Function types:

|  |  |  |
| --- | --- | --- |
| Name | Description | Notes |
| public | Anyone can call this function | Can only use one per function |
| private | Only this contract can call this function | Can only use one per function (either public or private) |
| view | Returns data without modification |  |
| constant | Returns data without moditfication | Same as view |
| pure | Will not modify or READ contract data |  |
| payable | Send ether to contract | Access and send ETH to contract |

Private functions are good for helper functions (only useful for within contract/ help other functions in contract)

**Remix Test Contract**

Contract -> compile to bytecode -> deploy instance on in-browser fake ethereal network (that’s rather handy)

Select **Run**

for **Environment**, select **JavaScript VM** (select in browser virtual ethereum network)

this creates a couple accounts in this environment with 100 ethernet

leave gas limit as is

value will be useful… later

drop down has contract that can be selected

select Inbox, note how string initialMessage is in deploy section… put some “text in this field”

After deployment, note how we have a selection of functions to call

Whenever a storage variable is marked as public, a new functions will automatically be created to view the data

**View vs Modify Functions**

Must Submit a transaction anytime one wants to change data, and wait for mining operation (data update time).

Asynchronous timing for data

Running Contract Functions

|  |
| --- |
| ‘Calling’ a Function |
| Cannot modify data |
| Can return data |
| Instant run |
| no cost to run |

Calling a function cannot modify contracts data, instantaneous and free to run

|  |
| --- |
| Sending a transaction to a function, “invoke” |
| Can modify a contract’s data |
| Takes time to execute |
| Returns the transaction hash |
| Costs money/gas to run |
|  |

Wei is small unit of Ether. There is no fractional unit of WEI. Smallest measure of Wei = 1 Wei

https://etherconverter.online

**Gas**

Transaction Costs for Ethereum Contracts

Gas system in ethereal is meant to measure how much work is required to execute the code.

Gas costs for operations can be found via online shared google doc

e.g. Add = 3 gas, multiply = 5 gas, added = 8 gas

Gastrice = amount of Wei sender is willing to per per unit of gas (price per gallon)

startGas/gasLimit = max units of gas the transaction can consume / willing to pay’

example with 14 needed

|  |  |
| --- | --- |
| gasPrice | 300 |
| gasLimit | 10 |

We we’re willing to pay for 10, but needed 14. The execution of the function will immediately halt at 10 units consumed, and further operations will stop.

|  |  |
| --- | --- |
| gasPrice | 300 |
| gasLimit | 20 |

Since we need 14, and are willing to pay for 20 gas units, all operations are executed. We don’t have to pay for the remaining 6 units (between 14 and 20).

In this example, we would pay: 300 Wei \* 14 = **4,200** wei, for the cost of the transaction.

**Redeploying Contracts**

X out contract on bottom right to close out instance

This video is a bit out dated for redeploying.

**What happens when we Deploy a Contract?**

New contracts have “to” field blank

“Data” compiled bytecode of the contract. All the code is visible to the world

**MetasMask multiple accounts**

Setting up multiple accounts to organize cash flow for operations / R&D / spending / staking (savings)

12 word pass phrase / mnemonic for metamask BIP39 mnemonic algorithm

iancoleman.io/bip39 mnemonic code converter

———————————————————

Programming Section

———————————————————

**Windows setup**  
Install:  
[Visual Studio Code](https://code.visualstudio.com/)  
[Node.js](https://nodejs.org/en/download/)  
[Git](https://git-scm.com/download/win)  
Powershell install as admin:  
npm install -g npm  
npm install -g windows-build-tools  
npm install -g ethereumjs-testrpc truffle  
  
Visual studio install litner:  
code --install-extension juanblanco.solidity

Truffle Test Framework  
<https://truffleframework.com/ganache>

Test ether:

faucet.rinkeby.io or just use **metamask Rinkeby “buy”**

Update node.js to version 8 or higher

Deployment options

Contract source -> solidity complier : [abi, contract bytecode]

Abi = works with javascript / api connection, bytecode goes to to ethereum network

**TRUFFLE** is a one stop shop for development of ethereum contracts

contract creation, local testing, and deployment

Problems with truffle: undergoing rapid development, some things just don’t work, stuff breaks… :/

of course, most ethereum tech is in infancy and likely to change. DEPENDENCIES will BREAK

… or we could create a custom node project to deploy contracts to Rinkeby

**Custom Node Project to Deploy Contracts**

- Setup a solidity complier

- Setup a custom mocha test runner that can test solidity

- setup script to complier and deploy contracts

Initiate a new json package

On terminal: npm init

Inbox project directory:

package.json

compile.js

deploy.js

contracts : inbox.solt

test: inbox.test.js

Solidity compiler via npm

Nom install —save solc

Don’t use require for inbox.sol with node (it’ll treat it as a javascript instead of solidity)

**Testing section**

Testing… we put it into the compiler and get the ABI and Bytecode

We need to write functional tests (test functions in contract

1. Compile > bytecode > deploy contract to local test network on machine **Ganache**/TestRPC

2. Compile > ABI > web3 > contract instance on Ganache

Installing mocha, ganache, and web3

npm install --save mocha ganache-cli web3

npm install --save mocha ganache-cli web3@1.0.0-beta.26 //older version used in these demos

**Web3**

Javascript app and ethereum connection

**Web3 Versioning issues:** v0.x.x and v1.x.x

V1 support for promises + async / await syntax (nice and pretty)

Beware of stackoverflow api calls for version0, as they are different using a lot of callback functions (hard to organize)

Web3 constructor function to make INSTANCES of web3 library

each instance is used to connect to an ethereum network (there can be multiple instances to connect to different test networks)

Web3 Starts web3 instance

Ganache -> provider (communication layer) -> web3

PersonA Email GroupB

We need to plug in a provider into web3

**Mocha**

A test running framework. Can test anytime of javascript code (front end/ back end) general purpose

Assertion = take two values… what are code produces and what i*t should have* produced

|  |  |
| --- | --- |
| Function | Purpose |
| **it** | Run a test and make an assertion |
| **describe** | Group together ‘it’ functions |
| **beforeEach** | Execute some general setup code |

Setup test functions, then **update package.json** scripts: {“test”: “mocha” // swap in the name mocha

npm run test // on command line to test

ISSUE WITH WEB3 not showing up, solve by command line: npm install ethereum/web3.js

Installs a better setup

**befroreEach** statement is used to execute some general initalitzatoin code for multiple tests.

any logic entered will run beforeEach IT block.

first block, initiates code in before each

second IT block, re initiates code from before each

if one sees the same setup code in each IT block, put it in beforeEach

beware of scope with const s in functions… so place the variable declaration outside

**Testing Contracts with Mocha**

Install Mocha globally with NPM:   
 npm install –global mocha

- deploy new contract to ganache test network (beforeEach = deploy a new contract)

- write some code to manipulate contract (manipulations will be slightly different for each assertions, so place in IT block)

- make an assertion for a value tied to the contract (assertions go inside it blocks)

- rinse and repeat

Accounts on Ganache

Web3 -> Ganache local test network [unlocked accounts]

web3 is asynchronous in nature. Functions return a promise… so we need a **.then or Async (which is better)**

Javascript Call Back Functions

- Passing Functions as Variables  
functions are “first class objects”, similar to strings, arrays, numbers..

function funcName(arg1, arg2){  
}  
function passthroughFunction (){  
 “I’m a function that will be used in another function”  
} // this is the callback variable!  
  
function baseFunction (x = passthroughFunction(){  
 console.log(x)  
}

[web3@1.0.0-beta26](mailto:web3@1.0.0-beta26)

inbox = await new web3.eth.Contract(JSON.parse(interface))

        .deploy({data: bytecode, arguments: ['Hi there!']})

        .send({from: accounts[0], gas: '1000000' });

Web3.eth.Contract(JSON.parse(interface)) // tells web3 to treat this object as a constructor  
  
.deploy({ data: bytecode, argruments: [ 1 to 1 mapping of arguments to array ] }) //

DEPLOY creates an object that CAN BE deployed, doesn’t actually deploy by itself  
SEND actually deploys it, asynchronously   
 from: person who is creating account, and gas amount  
  
Web3 can create a new contract, or interact with a contract that is already deployed.  
  
Requirements for working with contracts on network:  
**Interact with** deployed **contract**, need: **ABI, Address** of contract deployed  
**Create** a contract, need: **ABI, Bytecode**

Contract details… Methods object contains the functions for the contract  
 Options.address is the way to access the contract address