Getting started with Ethereum smart contracts.

“The Decentralized Internet”, “Web 3.0”, “DApps”, if these are some terms that got you curiosity and you already have made some initial efforts to understand them, this blog will help you get a better understanding of the fundamental unit that make these possible in the Ethereum blockchain (all notations will be in that regard)-‘Smart Contracts’. The following would be a beginner level discussion of how to write smart contracts and how they enable ‘Decentralized Applications’.

Lets start from a ‘Very High Level View’ of smart contracts.
In the Ethereum blockchain network there are two types of actors :

1. **Externally Owned Accounts (EOAs)**, accounts controlled by a public key-private key pair, (key_{public}, $key_{private}$). As of now, these account are held my humans!!

2. **Contract**, accounts controlled by code that is deposited in it. Code refers to a sequence of OpCodes (specific machine instructions represented a readable words) that does a predefined operation. For now stick to the idea that “Contracts contain code that can be executed”, more on that later.

Now lets see how these fits into the picture of a decentralized system which is provided by the blockchain and the consensus mechanism. Think of smart contracts as an “Automaton”. (If you haven’t seen one, try watching the movie Hugo (2011)). An automaton is mechanically programmed to do some very specific function when its keyed. Once an automaton is built for a functionality ‘**A**’, you can’t change it to functionality ‘**B**’. It just does what its programmed to, ‘**A**’.

Smart contracts are just that with an additional capability, it can persist data, which means it has a state (defined by the valued stored in it). They just does a set of functions which can be invoked from outside (EOAs) with data parameters. Now lets see how that would look like.

```
pragma solidity ^0.5.0;

contract  Registry {

        uint public registrySize;
        mapping (address=>int) registry;

        constructor() public {
                setMySecretNumber(int(0), address(0));
        }

        function getMyNumber() public view returns (int) {
                return registry[msg.sender];
        }

        function setMySecretNumber(int _mySecretNumber, address _address) public returns (uint)
        {
                registrySize++;
                require(_mySecretNumber != 0);
                require(registry[_address] == 0,'Item already exists!!');
                registry[_address] = _mySecretNumber;
                return registrySize;
```

```
    }

}
```

So this is a ‘Smart Contract’. If you have some familiarity with C++, Java... you would have already made a sense out of it. This is the Ethereum script called as ‘Solidity’. A Turing complete language that is a lot similar to C++ and javascript. let’s just dismantle the script.

1. pragma solidity ^0.5.0; pragmas are instruction to the compiler. The ‘version pragma’ here checks whether the designated compiler/set of compiler is to compile the contract, otherwise it throws an error. Since compiler version updates can have changes that might break your code, this is sort of an assurance that only specified compiler will get used. You can use comparators to specify a range of versions. Eg : pragma solidity >=0.4.22 <0.6.0;

2. The next segment of code the has a lot similarity with class definition is defining a contract called ‘Registry’. As said a contract consists of some state variables (which is ‘private’ by default) , a constructor and contract function definitions.
    1. uint registrySize; An unsigned integer of size 256. You can also specify it as uint256. Other than this Solidity has an entire set of primitive types. Some of the most used are int, string, bool, address. You can look up more here. All of the primitive types variables will be passed by value. Following the type you can specify two things,
        1. Visibility specifier : If you are from a C++ or Java world you got it, public and private. If the state variable is declared ‘public’ a getter function will get automatically generated upon compilation which can be used to access the state variable externally. Otherwise (private) its accessible to only that contract’s function, which is the behaviour by default. So ‘numberOfAddressess’ is private as nothing is specified.
        2. Storage specifier : All state variables are of type ‘storage’, which means they are persisted as the state of contract in the blockchain once deployed.

    2. mapping (address=>int) identityRegistry; Defines ‘registry’ as an associative array mapping type address to an signed integer. ‘address’ is a special type which can store 20 byte addresses, whether it be EOA or Contract address and as said its private by default.

    3. Then we have a constructor which a function call. Why add a zero address pointing to 0? In solidity all variables are give a default value that has meaning zero depending on its type ie, bool->false, string->””, int->0 ... .Just as a precautionary measure that nobody add such an invalid entry to the registry, we take care of that at the constructor.

    4. Now we have two functions, sort of a getter and setter for the registry. A function header is of the form, function <fnName>(<arg list>)[<visibility>] [<modifier>] [returns (type)] {}.
    visibility := public | private | external | internal
    modifier := pure | view | payable

5. You should be knowing whats meant by public and private by now, but what is external and internal ? These are two keywords that is in the realm of contracts and how they are designed,
    1. external -> these function can only be called from outside entities.No functions defined in the same contract can invoke them.
    2. internal -> the opposite of external. They can be invoked by only the same functions of the contract.

    We will see later on how these restrictions aid the idea of decentralization.
6. What are these modifiers?
    1. Pure -> a pure function can't modify or access the state.
    2. View ->a view function can't modify the state, by can access.
    3. Payable -> You show be aware of ether, to call the functions you should pay to the contract addresses. Does that make sense ? It will as we move on.
7. And what is that msg.sender ? It is a global variable denoting the address of whoever called the function whether it be a contract or EOA. This will become more clear as we built and test this contract.

Now that we have the syntactic overview of the entire contract lets see how this contract enables decentralization. Once you save this contract on a file with .sol extension, the next thing is to deploy it. Once thats done follow up these steps,

1. Setup 'Meta Mask', an ethereum client which you as an EOA can use to interact with the blockchain. You can check their YouTube channel to get more insight on this client and how it works.

2. Now that you have an account, copy you EOA address/ Account address, and use the Rinkeby Faucet to get some fake ethers for the Rinkeby test network. Post your Account address (Key_{public}) in any social network and paste the link to obtain fake ethers. Fake Ethers? Test Networks ? These are ethereum blockchains were you can deploy and test your Dapps spending fake ethers. If you choose to opt the 'main network', the real one it costs!!

3. Now that you have an account and have some fake ethers in Rinkeby test network, we will you the 'Remix IDE' to edit and deploy our contracts. Copy the code to the editor window and compile it (compiler versions available as dropdown).

4. Once you have compiled the smart contract without any errors, go to the 'Run' tab. Make sure that in MetaMask its the Rinkeby test Network and the current account is the one with the fake ethers. Choose the options in the 'Run' tab.
    1. Environment -> Injected Web3 (Rinkeby)
    2. GasLimit ? Seems familiar ? Whether it be execution of a contract or a deployment we need to pay Gas. Think it as sort of Gasoline needed to run the function/deployment.
        1. But who need this Gas? It the nodes in the blockchain network that would be execution the called function/ deploying a contract. We are paying the Gas in ethers "to run our requested computation in their machines".
        2. Even storing values to state variables costs gas. But thats anothers story that you can read up.

3. Why then a gasLimit? Thats the maximum amount of gas a called function can take for execution. It it runs out of gas at some point it return failing execution. Think of you calling infinitely running function. You can't basically because you need to pay an infinite amount of Gas, which is ether and thats finite in supply and costs !! The node thats executing your code decrements the gas supplied based on a predefined list of instructions. Once the gas gets to zero, the node stops execution and returns. If the called function gets successfully executed and still there is gas left (which can be zero also), the remaining gas is returned to the account which invoked the function.

3. Value -> If we want to sent ether (1 ether = 10^18 Wei) to a contract, set the value here. We will come back to it later.

5. Below choose the contract to be deployed (if the compiled file has multiple contract definitions) and click 'Deploy'. Thats it.., you will be prompted by MetaMask to confirm the contract deployment transaction paying some ethers as gas. If the chosen Account in MetaMask has ethers in it, accept the transaction. Else it will show a "Low Balance error".
6. ACCEPT the transaction and there we go, wait for a few seconds/minutes to get you deployed and you will get the 'deployed to' address back. In the Remix editor you will have the 'Registry' contract listed in the 'Deployed contracts' section.

Interacting with the deployed Registry contract.

Now you can watch the magic. Once the code is deployed, you can't alter its definition. It does what its defined as. The only thing EOAs and other contracts can do with it is 'interact to it with the public functions exposed'. So what happens here?

1. Hope you remember that you have an EOA. Copy that account address and set your secret number using setMySecretNumber().

2. Now try getMySecretNumber(). It should return the value you just set. Whats the magic here then? The value of you secret number is persisted to the blockchain using this contract and nobody else can retrieve it other than you (your account address) or alter it because the contract provide no function that does this. It can do only whats defined.

   If the same is being implemented as a centralized system, anybody who has access to the storage could possible alter your secret number or possibly steal it. Not the case with block chain, since your data is replicated across all (most) nodes in the network and any mutation invalidates the chain, nothing can be done. "The contract that only you access your secret number if enforced by the code"

3. How the registrySize() got a public getter function ? Told you, all public state variable gets one automatically!!

Contracts talking to each other !!

We saw how an EOA could talk to a deployed contract using the Remix IDE. Earlier in the discussion we saw contracts can communicate between each other also. Lets build a crowd-sale !!

Lets first understand what is it that we are building and why a decentralized version would be the better counter part.

1. Crowd-Sale : A sale where anybody can participate and buy the item offered by the entity organizing the sale. The item can be of inherent value or can represent shares of a company or establishment.

   In traditional systems the transaction, sale of 'X' items for 'Y' amount is recorded in documents and this pose problems such as trustworthiness of that piece of paper and need for blindly believing the authority thats signing its trustworthy, forging and alterations of sale document, lot of efforts to setup the documents and sale...

2. What happens when blockchain is being used for the same? All of this problems inherent to the traditional system gets solved by just the platform/network.

So how would a minimum viable design look like.

1. We should have a smart contract capable of,
    1. Receiving a amount and dispensing the sale item.
    2. Recording the sale.
    3. Once the sale is over it should credit the raised amount to the beneficiary of the sale.
    4. It should be able to close the sale on some conditions. (Items sold out)
2. What can be sold ? We can sell tokens, ERC 20 tokens!
   A token can be implemented very easily in the Ethereum blockchain using the ERC 20 Standard. You just have to define the standard interface. Have a look at it before we start.

Token Contract

//token code

There are three things here other than what we have already discussed,

1. Interface : This is all about contracts communicating with contracts. One way for communication is tho define an interface (abstract classes for C++ and interfaces for Java people). The interfaces need not define all it methods, only those which are to be accessed by the calling contract, like in this case the function 'receiveApproval(...)'

2. Inheritance : As with classes Contracts can also inherit, making the inheriting contract to access all of the inherited classes state variables and functions. Refer this for more info.

3. Library : are like smart contracts but are provisions for code resuse. Calling a function from a library executes the function in the context of the calling contract. Refer this for more info

4. Events : Events cause the arguments to be stored in a transaction log, a blockchain data structure.

5. ‘payable’ modifier: modifies a function or address such that it requires to pay ether, ie include value() to the transaction calling the function, while in the case for variables, address that are to be ‘paid to’ from a contract should be defined ‘payable’.

6. Type casting : refer this

Crowd Sale Contract