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 efforts to understand them, then this blog will help you get a better understanding of the fundamental unit that make these possible in the Ethereum blockchain -‘Smart Contracts’ (all notations will be in that regard). 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 accounts are held by humans!!

2. **Contracts**, 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 set of operations. 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 functions when its keyed. Once an automaton is built for a functionality ‘**A**’, you can’t change its functionality to ‘**B**’. It just does what its programmed to do, ‘**A**’. Like an automaton can be named, the blockchain assigns an address [1] to contracts. This smart contracts have one more feature, it can persist data, which means it has a state (defined the set of values its storage contain).

Now lets PICTURIZE !!
Automatons with memory sitting at certain nodes in a large network where each node is identified by a 20 byte address. The other nodes are occupied by EOAs (essentially humans), who build and interact with the automatons in the network.

There is one more catch, smart contracts are like ‘daemons’. Like the same daemon processes in a computer system, they are dormant, waking up when a ‘message call’ reaches them, executes whatever they are meant to do wrt to message and going dead again. So who sends this messages? Few lines back we discussed EOAs interact with smart contract. This is how they do, through ‘message calls’ which is basically a transaction. Is it only EOAs that can send messages? Nope, contract can in turn call other contracts also, but told you they are all dormant at first. So there should be an EOA to start the chain of message calls. Hope you guys got a picture. Now lets see how a smart contract 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’ to store your unique secretNumber. You don’t want anyone else to access/alter it and don’t want to change it once its set. Lets see how your secretNumber is safe with the contract. Before that lets understand the basic syntax. If you have some familiarity with C++, Java... you would have already made a sense out of it. It is written in the Ethereum script called as ‘Solidity’. A Turing complete language that is a lot similar to C++ and javascript.

Let’s just dismantle this script.

1. pragma solidity ^0.5.0; pragmas are instructions to the compiler. The ‘version pragma’ here checks whether the designated compiler/set of compilers 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(s) will get used. Here it means you code can be compiled with all subversions of 0.4.0. 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 definitions. It is defining a contract called ‘Registry’. As said a contract consists of state variables which are the store of the contract (visibility = private by default) , a constructor and contract function definitions. (It can also have events and user defined modifiers)
    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. As a data type is defined by the type and range of values it can have along with the list of operations possible, curious mind can have have a look here. All of the primitive typed variables are passed by value. As with data members of a class contract state variables also comes with two parameters,
        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 functions, which is the behaviour by default. So ‘register’ 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. You can checkout ‘memory’ and ‘callData’ storage specifiers here.

    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. ‘mapping’, ‘arrays’ and ‘structures’ are complex data types that are all passed as references.

3. Then we have a constructor with a function call. Why add a zero address pointing to 0? In solidity all variables are given a default value that has meaning zero (zero value) 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 what's 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, but can access them.
    3. Payable -> You should be aware of ether, to call these functions you should pay ethers to that contract addresses. Does that make sense? It will as we move on, so be patient.

7. And what is this msg.sender ? It is a global variable denoting the address of whoever called the function whether it be a contract or EOA. This way the caller of the getter function could retrieve only his secretNumber. This will become more clear as we build and test this contract.
8. require(boolean expression, ['explanation string']), is a way to check for some condition's validity mostly with input variables (function arguments). You can use assert(conditional exp) to enforce the same with state and local variables of a contract.
Both of these functions return from execution if the conditional argument evaluates to 'false'. Beware solidity doesn't consider empty items and zero as 'false' value as with many languages.

Now that we have the syntactic overview of the entire contract lets see how this contract enables decentralization. Once you saved this contract on a file with .sol extension say 'secretNumberContract.sol' the next thing is to 'deploy' it.

Contract Deployment !!

What secretNumberContract.sol is ? Just a solidity file containing a contract definition. Thats of no use unless its deployed on to a blockchain network. Its like designing a 'Bacteriophage' (virus to

kill harmful bacteria). It becomes useful once its injected to the body of the sick. Similarly a contract is only useful if its deployed. But what is deployment ? It is just a transaction carrying the compiled code for Ethereum Virtual Machine (EVM) with recipient addresses BLANK.
Wait, EVM ? Its essentially what that executes a contract code, as any machine is defined by its Instruction Set Architecture or ISA, this is a set of instructions (OpCodes) and their implementation which are to be used as the building unit for any execution logic.

But who would create this contract, and send it to the blockchain? As discussed in the first paragraphs 'EOAs build and interact with automatons', we need an external account to create the deployment transaction. Not just that we need some ethers to create this transaction, but why? The need to spend a part of some finitely available resource to do/request some work avoids the dumb machines from the 'Halting Problem'. Do give it a thought if you couldn't get your heads around first! Since we don't have any ethers or nor are we planing to spent actual money to get some and deploy this tutorial contract, let's just use a 'Test Network' where we get fake free ethers. So let's get these done one by one.

An EOA can be setup using an ethereum client. We will go for 'MetaMask', available as a browser extension,

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!!

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

3. 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? The nodes in the blockchain network that would be executing the called function/deploying the 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 another story that you can read up.
        3. Why then a gasLimit? Thats the maximum amount of gas a called function can take for execution. If it runs out of gas at some point it returns failing execution. Think of you calling function a function with an infinite loop. 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 (Gas per instruction). 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. That's good!!

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.

4. 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".

5. ACCEPT the transaction and there we go, wait for a few seconds/minutes to get it deployed and you will get the 'deployed to' address back. Thereafter in the Remix editor you will have the 'Registry' contract listed in the 'Deployed contracts' section.

Interacting with the deployed Registry contract.

The contract is now in the Rinkeby Test Network. 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() using the Remix IDE.

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 that reside in"

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 !! But let's 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 and the ease of setting up a contract agreement using smart contracts.

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 ‘item’ be sold ? We can sell tokens, ERC 20 tokens! Tokens are just like ethers, bitcoins, shares... only difference they can be designed as we want, just that it need to implement a basic interface defined by the ERC 20 standard. Have a look at it before we start.
3. But what is a token anyway? Like our register contract, it a smart contract with a initial supply of tokens (thats just a number), a data structure to keep track of who owns how much tokens, and function to facilitate transfer of tokens from one account to another, inquire about one’s balance etc. You might be getting the feeling it sounding like a ‘Bank dispatching a currency minted by itself’.In a way its a very basic form of ‘decentralized’ bank itself. The only difference is that the code defines and controls whatever functionality the contract would ever have. But lets see how this all would look like in code.

Token Contract

```
pragma solidity ^0.5.0;

contract ERC20Interface {

    function balanceOf(address _owner) public view returns (uint256 _balance);
    function transfer(address _to, uint256 _value) public returns (bool _success);
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool _success);
    function approve(address _spender, uint256 _value) public returns (bool _success);
    function allowance(address _owner, address _spender) public view returns (uint256 _remaining);


    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);
}

library SafeMath {
    function add(uint a, uint b) internal pure returns (uint) {
        uint  c = a + b;
        require(c >= a, "Safe Add assertion failed");
        return c;
    }

    function subtract(uint a, uint b) internal pure returns (uint) {
        require(b <= a, "Safe subtraction aseertion failed");
        uint c = a - b;
        return c;
    }

    function multiply(uint a, uint b) internal pure returns (uint) {
        uint c = a * b;
        require(a == 0 || c/a == b, "Safe multiplication assertion failed");
        return c;
    }

    function divide(uint a, uint b) internal pure returns (uint) {
```

```
        require(b > 0, "Safe division assertion error");
        uint c = a / b;
        return c;
    }
}

interface TokenRecipient {
    function receiveApproval(address _from, uint _value, address _token, bytes calldata _extraData) external;
}

contract Ownership {
    address public owner;
    address public newOwner;
    constructor() public {
        owner = msg.sender;
    }

    modifier ownerAction() {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address _newOwner) public ownerAction {
        owner = _newOwner;
    }

    function acceptOwnership() public {
        require(msg.sender == newOwner);
        emit ownershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = address(0);
    }

    event ownershipTransferred(address _from, address _to);

}


contract JTHCoin is ERC20Interface, Ownership {
    using SafeMath for uint;

    string public symbol;
    string public  name;
    uint8 public decimals;
    uint _totalSupply;

    mapping(address => uint) balances;
    mapping(address => mapping(address => uint)) allowed;

    constructor() public {
        symbol = "JTH";
        name = "JTHCoin";
        decimals = 18;
        _totalSupply = 1000000 * 10**uint(decimals);
        balances[owner] = _totalSupply;
        emit Transfer(address(0), owner, _totalSupply);
    }

    function totalSupply() public view returns (uint) {
        return _totalSupply.subtract(balances[address(0)]);
    }

    function balanceOf(address _tokenOwner) public view returns(uint) {
        return balances[_tokenOwner];
    }

    function _transfer(address _from, address _to, uint _tokens) internal {
        uint previous_balance = balances[_to] + balances[_from];

        balances[_from] = balances[_from].subtract(_tokens);
        balances[_to] = balances[_to].add(_tokens);

        emit Transfer(_from, _to, _tokens);

        assert(previous_balance == balances[_to] + balances[_from]);

    }

    function transfer(address _to, uint _tokens) public returns(bool _success) {
```

```
        _transfer(msg.sender, _to, _tokens);
        return true;
    }

    function approve(address _spender, uint _tokens) public returns(bool _success) {
        allowed[msg.sender][_spender] = _tokens;
        emit Approval(msg.sender, _spender, _tokens);
        return true;
    }

    function transferFrom(address _from, address _to, uint _tokens) public returns (bool _success) {
        allowed[_from][msg.sender] = allowed[_from][msg.sender].subtract(_tokens);
        _transfer(_from, _to, _tokens);
        emit Transfer(_from, _to, _tokens);
        return true;
    }

    function allowance(address _tokenOwner, address _spender) public view returns (uint _remaining) {
        return allowed[_tokenOwner][_spender];
    }

    function approveAndCall(address _spender, uint _value, bytes memory _extraData) public returns (bool _success) {
        TokenRecipient spender = TokenRecipient(_spender);
        if(approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, address(this), _extraData);
            return true;
        }
    }

    function () external payable {
        revert();
    }

}
```

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 reuse. 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

The transferFrom(), approve() and approveAndCall() allows this token to connect with crowd sale contract.

Crowd Sale Contract

```
pragma solidity ^0.5.0;

interface JTHCoin {
    function transfer(address _receiver, uint value);
}

contract CrowdSale {
    address public beneficiary;
    address public tokenAddress;
    uint public tokensPerEther;
    uint public minEtherAccepted;
    uint public maxEtherAccepted;
    mapping(address => uint) registry;
    uint public startedAt;
    uint public endsAt;
    uint public saleValue;
    uint public saleGoal;
    uint public participantCount;
    uint public bonousPercentage;

   constructor(
        address _beneficiary,
        address _tokenAddress,
        uint _tokensPerEther,
        uint _SaleTimeSpanInSeconds,
        uint _minEtherAccepted,
        uint _maxEtherAccepted,
        uint _tokensAvailable,
        uint _tokenPriceInEther,
        uint _bonousPercentage) public {

        beneficiary = _beneficiary;
        tokenAddress = token(_tokenAddress);
        tokensPerEther = _tokensPerEther;
        saleGoal = _tokensAvailable * _tokenPriceInEther;
        startedAt = now;
        endsAt = now + _SaleTimeSpanInSeconds;
        minEtherAccepted = _minEtherAccepted;
        maxEtherAccepted = _maxEtherAccepted;
        bonousPercentage = _bonousPercentage;
        saleValue = 0;

        emit CrowdSaleTimeLine(address(this), startedAt, endsAt);
    }

    event CrowdSaleTimeLine(address saleContract, uint startedTimeSinceEpoch, uint endingTimeSinceEpoch);
    event SaleGoalReached(uint now);


    modifier eventLive() {
        require(now <= endsAt, "Sale Ended");
        _;
    }

    function saleGoalNotReached() internal eventLive returns (bool) {
        if(saleValue < saleGoal)
            return true;
        else
            return false;
    }

    function () payable external {
        require(saleGoalNotReached());
        uint amount = msg.value;
        uint ntokens = tokensPerEther * amount;

        if(++participantCount <= 10) {
            ntokens += ntokens * bonousPercentage;
        }

        tokenAddress.transfer(msg.sender, ntokens);

    }
    }
}
```

Before discussing any thing lets deploy these and see them in action. Deploy the token first, then you will be getting an address of the token contract. Deploy The crowd-sale contract and run its construct with the token address you just got.

Once reading through the crowd you would have made sense of most of code. The one specific thing is how we are communicating.

1. Token contract has some 'totalSupply', crowd-sale contract needs to access it. We know now these tokens belong to who ever deployed the contract, thats your EOA. So lets use the magic function approveAndCall() to approve the address = 'crowd-sale contract' address to transfer the coins from your EOA to its account, so that the contract manages the sale. But how?
    1. Thats where the functions approve(), and transferFrom() from comes in. These sets information in the 'allowed' mapping that your account allowed this address the authority to transfer atmost nTokens on your behalf. The transferFrom() implements this. That seems fine, by where is the definition for function receiveApproval() called inside approveAndCall(). So this is a function defined in the crowd-sale contract and we use the interface and the address of the crowd-sale contract in token contract to do the message call !! This is one way to communicate with another contract!! You just need to know the address of the contract and create and interface for 'functions to be called' at that interface.

    2. So try calling approveAndCall() using the contract address. You can check the balance of crowd sale contract now and see the magic. Now play on with the crowd sale till the time span ends. Use MetaMask to send ethers to crowd-sale contract address.

Concluding

You can play with lot more of contracts at the 'Solidity by example' page and checkout the 'solidity in depth' section to get better insights. Wait, thats all!! No, what we did here was to use solidity to create contracts and deploy and test them using remix, that wont make a DApp. You need a setup to deploy, test and interact with the contracts programatically. Checkout the available clients and framework 'truffle'+'ganache'+'drizzile' and 'web3.js'.

Still not okay with smart contracts, go here and build a 'Governence' and extend the 'crowd-sale'

Okay, one last thing to ponder about, the theory of 'Emergence'.

Do ponder !!

Wish you the very best with smart contract and let see them 'emerge'!!