Building a Basic Ethereum Smart Contract (For Bike Park Fun)

Author	josh
Date	2020-06-26 11:54:07

Overview

Ethereum smart contracts offer a wealth of possibilities for building decentralized, blockchain-secured applications. But what is a smart contract? These programs are code executed by the Ethereum Virtual Machine, a decentralized computer on the Ethereum network.

Every node on the network receives the contract code, and validates every function call sent through an Ethereum transaction. After the function executes, the new *state* of that contract is stored on the blockchain for everyone else to verify.

There's a lot of potential here, but understanding how smart contract development differs from traditional programming can be a tad daunting. Let's talk about the basics and build our very own smart contract for something fun!

Understanding Smart Contract Development

Smart Contract Properties

Smart contract code has two very important properties that differ somewhat from traditional development using a language such as C, C++, or Python.

First, smart contract code is *deterministic*. Since we are writing code for a decentalized, global network of nodes that will all run the same code and expect the same *result*, our code must have this property of determinism. Every time a smart contract runs on a node, another node running that code will get the *exact same result*.

Smart contracts are also *atomic.* This means that the function either fully completes successfully and the state changes are *committed* to the blockchain. If there's any sort of error in execution (such as running out of gas, or another error), then the changes are *completely rolled back and not committed.* There's never any intermediate state change left by a function execution - it either works in full or it doesn't.

These properties ensure that our network can properly operate in a decentralized manner. Every node, given the smart contract code and a transaction, will verify the same result to achieve network consensus.

Functions, Deploying, and Calling

In order to write an Ethereum contract, one will almost universally use a higher level language such as Solidity. Solidity code is then compiled into *EVM bytecode* - this intermediate language is understood by the *Ethereum Virtual Machine* that runs on network nodes.

The EVM bytecode is sent in a special transaction to the *zero address,* which deploys the contract on the blockchain and generates a unique *contract address* which the contract will live at. Any future function calls/transactions regarding this contract will be made to this address.

Now, there are two ways which smart contract functions can actually be called. The first is a local call to a node without making a transaction, using an API such as Web3.js. This call immediately executes and shows a return value, which is very useful for debugging and view-only functions that *do not change the contract's state on the blockchain.*

If there's a need to change some blockchain state with a function call, then a transaction must be use. This is our second method of executing a function and obtaining return data. By creating an Ethereum transaction to the contract (with gas payment for execution), the tx will be included in a block and the appropriate state will be update on chain when the block is mined.

Return data is a bit more complex using transactions over local calls. Transactions are inherently *asynchronous* operations. A transaction with a call is created, but it doesn't *complete* until it is included in an Ethereum block. Blocks on average take about 15 seconds to process, which is far too long to hold up front end code for. That, and the function could error out too!

So we deal with this by using smart contract *Events.* If your function needs to return a value to calls via transaction, you create an event. The event is triggered from within your function at the appropriate spot. Your DAPP frontend code then *listens for the Event*, so when the transaction is complete and included in a block, then frontend can retrieve the return value from the event.

An Actual Example - Bike Park Dice

What does a backyard bike park have to do with Ethereum smart contracts? Well, nothing really. But I like my coding projects to be fun and engaging, so I often tie them into other interests and hobbies that I have. So for this example, I'm creating a pseudo-random number generator contract that will help me pick mountain biking features to practice on in my yard. Why not, learning about blockchain should be fun!

Our smart contract code is short and sweet, and looks like this (with comments):

```
pragma solidity ^0.6.6;

contract BikeParkDice
{
    // The only function for this contract
    // It uses some seed data and a hash to return a "random" number from 1-6
    function rollDice() public view returns(uint256 final_roll)
    {

        // Get the sending address, block difficulty, and block timestamp
        // to give us a good enough "seed" for our pseudo-random number
        bytes32 seed = keccak256(abi.encodePacked(msg.sender));
        bytes32 seed_2 = keccak256(abi.encodePacked(block.difficulty));
        bytes32 seed_3 = keccak256(abi.encodePacked(block.timestamp));
```

Let's discuss the first half of our code here. This project present a tad unique challenge. Remember that smart contract code must be *deterministic*, so we certainly cannot generate truly random numbers!

Instead we will be using some "seed" data from the transaction that will be fairly unique, and using a hash function to create the appearance of pseudo-randomness.

In the code above, we first declare our contract and a function that will return an integer number. This function is also a *view* meaning that it does not change the state of the contract on the blockchain.

We take "seed" data from the msg.sender (function calling address - either a normal externally owned account, or another contract), the block difficulty, and block timestamp. This provides sufficiently "different" data for every call such that we'll get an appearance of randomness.

The abi.encodePacked simply helps us deal with data types - Solidity is a type language, and packing then hashing our seed means we can easily pass the seed data to our final hash.

```
        // Hash the seed data. This keccak hash is deterministic (necessary
for Ethereum contracts!)
        // and is preimage resistant, which is helpful for our pseudo-random
generation
        bytes32 hash = keccak256(abi.encodePacked(seed, seed_2, seed_3));

        // Cast the hash bytes to an integer value
        uint256 hash_num = uint256(hash);

        // Return a number from 1-6 by using the modulo (remainder) function
        final_roll = (hash_num % 6) + 1;
        return final_roll;

    }
}
```

Next, we take all of our seed data and run it through keccak256 (SHA-3), a cryptographically secure hash function. This helps our pseudo-randomness some more, as these kinds of hash functions are *preimage resistant.* There's no way to tell what the output might look like for a given input. They are also, very importantly, deterministic. Every node that validates a transaction with the same msg.sender, timestamp, and difficulty will get the same "random" roll.

Finally, the return of the 256 bit hash is a much larger number than we want. So we are using the modulo function to narrow our random range down to six. Modulo will take the hash value divided by six and give the *remainder*, from 0-5. We then add 1 to avoid an off-by-one error and get our 1-6 final output.

To test this, I compiled and deployed using the free and open source Remix IDE, an online development environment for solidity. It features a JavaScript VM blockchain for testing code without having to deploy to Ethereum or Ethereum Testnet for real.

In our case since this is a *view* (read-only) function, I could just do some test calls to the VM to get some sample outputs! I got 1, 2, and 4 in a few rolls.

Ethereum Smart Contracts - Solidity and MTB fun!

The final result here is a simple, but useful smart contract! This code compiles error free and I was able to test using a virtual blockchain environment. In a future iteration, I could add events and test on the real Ethereum blockchain, but this local-call version serves our purposes for learning the basics.

Again, I encourage anyone learning a new coding skill to have fun with it! You don't just have to follow a tutorial - think about a project you could use for another hobby or interest of yours . For me, I used this to randomly pick some mountain biking features in my yard to practice with. Why not? sending it with Solidity is fun!

One of the features my contract picked for practice - a North shore style roll-off