Programming blockchain part 2: Javascript programming language

Along with HTML and CSS, it is one of three core technologies in World Wide Web Content Production. Javascript is often used to create highly interactive websites.

1. Programming blockchain part 1: C ++ programming language

In the first part you learned about the problems that blockchain developers will encounter when conducting blockchain programming, as well as knowing why C ++ is chosen as the base language of Bitcoin source code.

In this section, you will learn about JavaScript, how to create a simple blockchain using JavaScript.

Javascript programming language

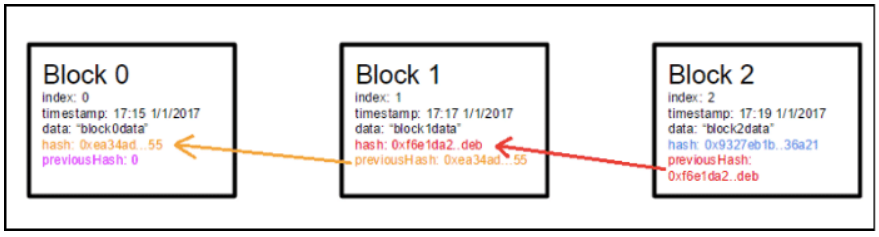
Next we will switch to Javascript.

Along with HTML and CSS, it is one of three core technologies in **World Wide Web Content Production** . Javascript is often used to create highly interactive websites. So now, this article will help readers find out how to create a simple blockchain using Javascript.

Suppose, we want to create a simple blockchain in Javascript. Before doing this, there are some things that we need to clarify.

What is Blockchain and exactly how does it work?

Blockchain is basically a series of blocks containing data. It is basically a linked list. So what makes it so special? A blockchain is immutable. When a data enters a block, it can never be changed. How does a blockchain achieve immutability? It is due to a simple but ingenious mechanism called "hashing". See the diagram below:



Each block is connected to the previous block through a hash pointer containing the hash of the previous block. One of the most compelling attributes of cryptographic hash is that if you change the input even a little, it can

greatly affect the output hash result. For example, you can try checking this out:

INPUT	HASH
This is a test	C7BE1ED902FB8DD4D48997C6452F5D7E509FBCDBE2808B16BCF4EDCE4C07D14E
this is a test	2E99758548972A8E8822AD47FA1017FF72F06F3FF6A016851F45C398732BC50C

Just converting "T" to capitalization will often change the result hash significantly.

So, how does this affect blockchain?

Each block is connected to the previous block via the hash pointer. So if someone has spoofed data in a block, it will change the hash a lot and the end result is affecting the entire string (because all blocks are linked). This will freeze that string and from here the blocks cannot be changed.

So how do we create a block? What does a simple block include? In simple **cryptocoin** (digital money) we will create (temporarily called " **BlockGeeksCoin** "), each block will have the following information:

1. Index: To know the block number.
2. Timestamp: To know the creation time.
3. Data: Data inside the block.
4. Previous Hash: Hash of the previous block.
5. Hash: Hash of the current block.

Before continuing, you need to understand some of the following terms:

1. This: The " **this** " keyword is called inside a function and allows you to access values ??within a specific object called that function.
2. Constructor: Constructor is a special function that can help create and initialize an object in a class. Each class is limited to only one Constructor.

Now everything is done, start creating blocks.

Create blocks

```
const SHA256 = require ("crypto-js / sha256");
Block class
{
constructor (index, timestamp, data, previousHash = '')
{
this.index = index;
this.previousHash = previousHash;
this.timestamp = timestamp;
this.data = data;
this.hash = this.calculateHash ();
}
calculateHash ()
{
return SHA256 (this.index + this.previousHash + this.timestamp + JSON.stringify
}
```

```
}
```

Analyze the code

The first line of the above code is used to call the **crypto-js** library because the **hash sha256** function is not available in JavaScript.

Next, call a constructor inside the class to call objects with certain values. The first thing you see is the **calculateHash ()** function. Let's see exactly what function it is performing.

In a block, we will get all the contents and their hash to get the hash of that particular block. Use **JSON.stringify** function to turn block data into string.

So we have successfully created the block. Now connect the blocks together into a blockchain.

Create blockchain

```
Blockchain class
{
// Section 1 Genesis block creation
constructor ()
{
this.chain = [this.createGenesisBlock ()];
}
createGenesisBlock ()
{
return new Block (0, "01/01/2017", "Genesis block", "0");
}
// section 2 adding new blocks
getLatestBlock ()
{
return this.chain [this.chain.length - 1];
}
addBlock (newBlock) {
newBlock.previousHash = this.getLatestBlock (). hash;
newBlock.hash = newBlock.calculateHash ();
this.chain.push (newBlock);
}
// section 3 validating the chain
isChainValid ()
{
for (let i = 1; i
{
const currentBlock = this.chain [i];
const previousBlock = this.chain [i - 1];
if (currentBlock.hash! == currentBlock.calculateHash ()) {
return false;
}
if (currentBlock.previousHash! == previousBlock.hash)
{
return false;
}
```

```
}
return true;
}
}
```

Analyze the code

There are many things to analyze in the code above. Break it down into sections.

Part 1: Genesis block

What is the genesis block?

The genesis block is the first block of the blockchain, and the reason why it is especially because while each block points to the previous block, the genesis block does not point to anything before. Therefore, the moment a new sequence is created, the genesis block is called immediately. In addition, you can see the ' **createGenesisBlock ()** ' function, in which block data is provided manually:

```
createGenesisBlock ()
{
return new Block (0, '01 / 01/2017 ',' Genesis block ',' 0 ');
}
```

Now, the genesis block has been built. Let's build the rest of the string.

Part 2: Add blocks

First, we need to know what the last block in the current blockchain is using the **getLatestBlock ()** function.

```
getLatestBlock ()
{
return this.chain [this.chain.length - 1];
}
```

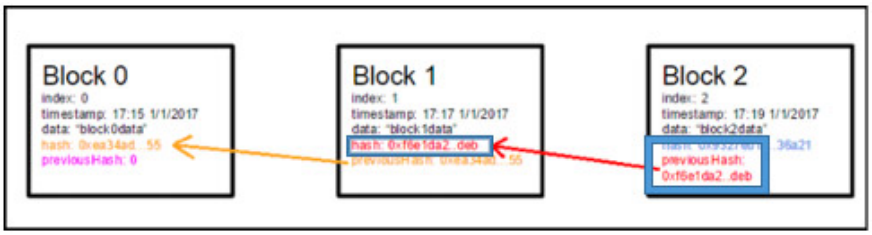
Now define the latest block. Find out how to add new blocks!

```
addBlock (newBlock) {
newBlock.previousHash = this.getLatestBlock (). hash;
newBlock.hash = newBlock.calculateHash ();
this.chain.push (newBlock);
}
```

So how do I add blocks? How to check if a given block is valid?

Do you still remember the content of a block and a block that has the hash of the previous block?

What we will do here is very simple. Compare the **previousHash** value of the new block with the hash value of the nearest block.



If these two values ??match, then this means the new block is valid and it has been added to the blockchain.

Part 3: Validate the string

Now, we need to check if there is a problem with the blockchain created and make sure everything is stable.

We need to use the " **for** " loop to loop from block 1 to the last block. Genesis block is block 0.

```
for (let i = 1; i
{
const currentBlock = this.chain [i];
const previousBlock = this.chain [i - 1];
```

In this part of the code, we define two terms, the current and the previous block. And now we just need to find the hash of these two values.

```
if (currentBlock.hash! == currentBlock.calculateHash ()) {
return false;
}
if (currentBlock.previousHash! == previousBlock.hash)
{
return false;
}
}
return true;
}
```

If the " **previousHash** " of the current block is not equal to the " **Hash** " of the previous block, this function will return **False** , otherwise it will return **True** .

Use blockchain

Now, use blockchain to create BlockGeeksCoin.

1. Set BlockGeeksCoin = new Blockchain ();
2. BlockGeeksCoin.addBlock (new Block (1, '20 / 07/2017 ', {amount: 4}));
3. BlockGeeksCoin.addBlock (new Block (2, '20 / 07/2017 ', {amount: 8}));

We have created a new cryptocurrency based on the blockchain and named it BlockGeeksCoin. By calling this new object, the constructor will be activated and in turn create the automatic Genesis block.

Just add two more blocks to the blockchain and give it some data. This is quite simple.

There's more.