Blockchain Node.js Introduction IV

Building a Blockchain with Node.js

Blockchains are meant to be decentralized and secure on a P2P network, so we'll need to implement a hashing algorithm.

Why hashing over other security methods?

Hashing allows us to input any size data and always get in return a string of a determined length, thus making it very easy to work with our data since the same input will always return the same output.

It is also nearly impossible to get back to an input with an output using a hash.

We'll be using the SHA-256 hash for this blockchain, provided to us by the crypto-js module: npm install --save crypto-js

Building the block class

A block in our blockchain will store an index, a timestamp, some data, the hash of the previous block, the block's own hash, and a nonce value, which defines the "work" that miners do to honor the set difficulty.

Create a blockchain.js file and start by making a constructor for the Block class in your chain.

```
class Block {

  constructor(index, timestamp, data, previousHash = "") {
    this.index = index;
    this.timestamp = timestamp;
    this.data = data;

    this.previousHash = previousHash;
    this.hash = this.calculateHash();

    this.nonce = 0;
```

```
  }

  /// our functions...

}
```

If you would like to get the current timestamp instead of manually inputting the timestamp, you can specify this by using JavaScript's Date() API above the Block class.

```
var dt = new Date();
var timestamp = dt.toString();
```

Hashing

Next, we'll define the calculateHash() function in this class, but first we need to import our SHA256 function from the crypto-js library on the first line of our file.

```
const SHA256 = require("crypto-js/sha256");
```

Now, we can hash our block data.

As used in the constructor, we'll define the calculateHash() function right below, which will input the index, timestamp, data, previous hash, and the nonce value to the SHA256() function and return the hash as a string.

```
calculateHash() {

   return SHA256(this.index + this.previousHash + this.timestamp +
JSON.stringify(this.data) + this.nonce).toString();

  }
```

Great! We now have a basic block constructor.

Next we'll have to create a constructor for the chain itself.

Blockchain constructor

To keep our blockchain simple, we'll be storing our blocks in an array called chain.

```
class Blockchain {

  constructor() {
    this.chain = [this.createGenesis()];
  }

  // our functions...

}
```

All blockchains have an initial block, known as the genesis block.

We will also be creating this genesis block with a createGenesis() function and put it at index 0 in our array.

```
createGenesis() {
    return new Block(0, "01/01/2018", "Genesis block", "0");
}
```

For our genesis block, we have provided an index of 0, a random timestamp, a ”Genesis block” string as its data, and ”0” as a random previous hash since there is no actual previous block.

Adding blocks

Since blockchains are meant to be composed of multiple blocks, we must add a latestBlock() and addBlock() function in our Blockchain class so we can actually form our chain.

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

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

With these functions, latestBlock() will retrieve the most recently created block that addBlock() will use to retrieve the previous hash.

addBlock() will then push the new block into the chain array.

Validation

Since our blockchain is not actually decentralized on a P2P network, we'll be implementing a simple validation function that makes sure all of our blocks are legit.

This runs a 'for' loop, checking, by index, the blocks in the chain array by recalculating the hash and comparing it to the stored hash of the block.

If this returns true, it then checks whether the previous hash stored in the block matches the hash of the block before the current block.

```
checkValid() {

    for(let i = 1; i < this.chain.length; 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;

}
```

Mining

Mining is one method used to make blockchains more secure, especially in the case of cryptocurrency.

It allows implementing a proof of work protocol, which checks the legitimacy of the block mined and only validates once a consensus is reached by the nodes on the P2P network.

Mining also controls the flow of blocks into the chain; increasing the difficulty will increase the time needed to successfully mine a block.

Since we will not be using a P2P network for our blockchain, the mining aspect will simply affect the time needed to add a block to the chain, no consensus needed.

Let's go back to the Block class and add a mineBlock() function under the calculateHash() function.

To make it more difficult for our computer to create a block, we can define how a block's hash should be.

In this case, we'll be telling our computer to figure out a hash of a block beginning with a specific amount of 0's.

```
mineBlock(difficulty) {
    while(this.hash.substring(0, difficulty) !== Array(difficulty + 1).join("0")){
```

```
      this.nonce++;
      this.hash = this.calculateHash();
    }
    console.log("Block mined: " + this.hash);
  }
```

Our while loop takes a substring of a calculated hash, from index 0 to the index of the given value of the difficulty, and continues to run if this substring does not equal an array of 0's that is the length of the given difficulty.

And since this loop will have multiple instances, we create a nonce value that changes every instance.

To set the difficulty, let's add the difficulty to our Blockchain constructor.

(Note that it will take much longer to mine a block with a difficulty greater than 5. For testing purposes, I recommend leaving it at 4.)

```
class Blockchain {
  constructor() {
    this.chain = [this.createGenesis()];
    this.difficulty = 4;
  }
```

And then to actually implement this, let's add mineBlock() to the addBlock() function:

```
addBlock(newBlock) {
 newBlock.previousHash = this.latestBlock().hash;
 newBlock.mineBlock(this.difficulty);
 this.chain.push(newBlock);
}
```

Testing

We will be testing our blockchain in terminal.

Add the following logs at the end so we can output the data to the terminal.

You may put whatever you'd like for the new Block parameters.

The following will also return the validity of your blockchain.

(Note that the parameters must be included in the order of index, timestamp, and data.)

```
let testChain = new Blockchain();

console.log("Mining block...");

testChain.addBlock(new Block(1, timestamp, "This is block 1"));

console.log("Mining block...");

testChain.addBlock(new Block(2, timestamp, "This is block 2"));

console.log(JSON.stringify(testChain, null, 4));

console.log("Is blockchain valid?" + testChain.checkValid().toString());
```

cd into your project folder and run node blockchain.js.

The program should output something like this:

```
Mining block...
Block mined:
000057d0ca3e051c2d3ce596d6c31941bd6734ad47a95100d8ab08871c9e9016
Mining block...
```

Block mined:
0000f68bfebb8a68e40da050688ba487dc53820b272b7945f17922058c40a928


```
{
   "chain": [
      {
         "index": 0,
         "timestamp": "01/01/2018",
         "data": "Genesis block",
         "previousHash": "0",
         "hash":
"a5d991a6c75bc497e967f6577445953d6a68e91a4243bbefad61f545f236045b",
         "nonce": 0
      },
      {
         "index": 1,
         "timestamp": "Mon Jun 25 2018 15:03:51 GMT-0400 (Eastern Daylight Time)",
         "data": "This is block 1",
         "previousHash":
"a5d991a6c75bc497e967f6577445953d6a68e91a4243bbefad61f545f236045b",
         "hash":
"000057d0ca3e051c2d3ce596d6c31941bd6734ad47a95100d8ab08871c9e9016",
         "nonce": 147287
      },
      {
         "index": 2,
         "timestamp": "Mon Jun 25 2018 15:03:51 GMT-0400 (Eastern Daylight Time)",
         "data": "This is block 2",
         "previousHash":
"000057d0ca3e051c2d3ce596d6c31941bd6734ad47a95100d8ab08871c9e9016",
         "hash":
"0000f68bfebb8a68e40da050688ba487dc53820b272b7945f17922058c40a928",
         "nonce": 56240
      }
   ],
   "difficulty": 4
}
```


Is blockchain valid? true

There you have it!

Try making a ‘for’ loop to generate a set amount of blocks instead of having to hard program it.