**Blockchain Node.js Introduction II**

**Building your own tiny blockchain in JavaScript**

Beyond the social, financial, and technological disruptions that blockchain technology has the power to create, the simple idea that a piece of code could be behind all this has always is fascinating.

Let's take a look at how we'd build a very simple block blockchain to help visualize how transactions are added.

Requirements:

This app is built in Node.js.

We'll also be making use of the sha256 npm package to hash the blocks.

If you're new to Node.js, first create a new directory, create an index.js file, and then from your terminal, run npm init -y from within your new directory. This creates a package.json file, which will allow us to install the sha256 package.

Run npm i sha256 to install the sha256 library.

If you're on Repl.it, create a new Nodejs repl, install thejs-sha256 package, and require it.

```
const sha256 = require('sha256');


class Block {
  constructor(index, timestamp, data, prevHash) {
    this.index = index;
    this.timestamp = timestamp;
    this.data = data;
```

```
    this.prevHash = prevHash;
    this.thisHash = sha256(
      this.index + this.timestamp + this.data + this.prevHash
    );
  }
}
```

Now that we're set up, it's time to create a new class so we have a template that we'll use to build each block.

Each block on our simple blockchain will contain 5 pieces of data: the block's index, the precise time it was created, some arbitrary data (for bitcoin, this data would be the addresses of the sender and receiver of bitcoin and the amount of the transaction), the hash value of the block preceding it, and a hash of itself.

A hash is a digital fingerprint of a piece of data. We'll see later how this part of each block makes it extremely difficult for anyone to alter the blockchain once a block has been confirmed.

To create our Block, I'm using a JavaScript class, but a constructor function would work just as well since that's really all a class is under the hood.

I set the constructor function to take all of our data except for the hash as parameters.

The block will then look at the index, timestamp, data, and previous hash, and generate its own hash or fingerprint.

You may have noticed that since each block requires the hash of the previous block, we have a problem of how to create the first block.

The first block, known as the Genesis Block, will get handed zeros as the previous hash, so we'll have to write a function to create that single block.

```
const sha256 = require('sha256');


class Block {
  constructor(index, timestamp, data, prevHash) {
    this.index = index;
    this.timestamp = timestamp;
    this.data = data;
    this.prevHash = prevHash;
    this.thisHash = sha256(
      this.index + this.timestamp + this.data + this.prevHash
    );
  }
}


const createGenesisBlock = () => new Block(0, Date.now(), 'Genesis Block', '0');
```

On line 15, I use an arrow function to create a new block based on the class we wrote earlier. I hand it a 0 for the index, the current time, a string stating that it's the Genesis Block, and a previous hash of 0.

We now have our first block, and we're almost done! The genesis block is special, but every other block will use the same formula going forward.

```
const sha256 = require('sha256');
```

```
class Block {
  constructor(index, timestamp, data, prevHash) {
    this.index = index;
    this.timestamp = timestamp;
    this.data = data;
    this.prevHash = prevHash;
    this.thisHash = sha256(
      this.index + this.timestamp + this.data + this.prevHash
    );
  }
}

const createGenesisBlock = () => new Block(0, Date.now(), 'Genesis Block', '0');

const nextBlock = (lastBlock, data) =>
  new Block(lastBlock.index + 1, Date.now(), data, lastBlock.thisHash);
```

Here, I write a nextBlock function that takes only the last block, and the new block's data as parameters. The function then creates a new block and sets the index of the new block to whatever the previous block's index was and adds 1, grabs the current time, and then grabs the previous block's hash and sets it as prevHash on the new block.

This last step, while only a tiny bit of code, is extremely important. This links each block to its previous block through a hash.

Let’s say someone wanted to alter the bitcoin blockchain so that it looked like they had received more bitcoin than they actually did, or to make it look like they spent less of their bitcoin.

If they altered a block, the hash, which serves as a snapshot of the data the block contained when it was created, would no longer match. To remedy this, they’d have to rehash the block. Since all the blocks are linked together and point back to the previous block’s hash, this person would also have to alter all the blocks on the chain following their target block in order to make it look legitimate.

While modern computers could conceivably alter a block, generate new hash, and then generate new hashes for all the following blocks so that the altered block was seen as legitimate, bitcoin in particular implements a method called ‘Proof of Work’ to slow down the confirmation of new blocks, making it nearly impossible.

Also, due to the distributed network of ledgers that a blockchain relies on, a nefarious actor would have to control more than 50% of the network in order for their counterfeit blockchain to be accepted.

Our blockchain won’t be distributed or utilize proof of work, but it’s an extremely clever solution to the problem and those topics warrant their own explanations.

So now we have everything we need to generate our tiny, simple blockchain!

Let’s write a function that will run as many times as we want, adding new blocks, and then we’ll take a look at what it’s created.

```
const sha256 = require('sha256');


class Block {
  constructor(index, timestamp, data, prevHash) {
    this.index = index;
    this.timestamp = timestamp;
```

```
    this.data = data;
    this.prevHash = prevHash;
    this.thisHash = sha256(
      this.index + this.timestamp + this.data + this.prevHash
    );
  }
}

const createGenesisBlock = () => new Block(0, Date.now(), 'Genesis Block', '0');

const nextBlock = (lastBlock, data) =>
  new Block(lastBlock.index + 1, Date.now(), data, lastBlock.thisHash);

const createBlockchain = num => {
  const blockchain = [createGenesisBlock()];
  let previousBlock = blockchain[0];

  for (let i = 1; i < num; i += 1) {
    const blockToAdd = nextBlock(previousBlock, `This is block #${i}`);
    blockchain.push(blockToAdd);
    previousBlock = blockToAdd;
  }
  console.log(blockchain);
};
```

```
const lengthToCreate = 20;

createBlockchain(lengthToCreate);
```

On line 20, i create a new function that will create a new blockchain at any length you specify.

I create a new blockchain that we'll store in an array, and I place the genesis block as the first element in the array.

Next, I create each block one at a time, handing it the data it requires from the previous block, and then I place that new block into our array.

Once we're done, I log out the result. Let's take a look at the first few blocks to see what we've created.

```
{
   index: 0,
   timestamp: 1537236800416,
   data: 'Genesis Block',
   prevHash: '0',
   thisHash: 'a0fec317fe451b2aaeefc62db8e58a69a2eb72811bc0618693815c428882e4b0'
 },
 {
   index: 1,
   timestamp: 1537236800417,
   data: 'This is block #1',
```

```
  prevHash:
    'a0fec317fe451b2aaeefc62db8e58a69a2eb72811bc0618693815c428882e4b0',
  thisHash: 'ef4c1756bb4773185d96164d29c05ed1f9bb45693fb6a85748c694b470c911fb'
 },
 {
  index: 2,
  timestamp: 1537236800417,
  data: 'This is block #2',
  prevHash:
    'ef4c1756bb4773185d96164d29c05ed1f9bb45693fb6a85748c694b470c911fb',
  thisHash: '9a811c23b4a8bc6bed733a160936053b2ade9920878b57f7877a99ae590d7f87'
 },
 {
  index: 3,
  timestamp: 1537236800417,
  data: 'This is block #3',
  prevHash:
    '9a811c23b4a8bc6bed733a160936053b2ade9920878b57f7877a99ae590d7f87',
  thisHash: '3440c7ebf00d9dd856c4cc2adb79863d803ec821b4083fda7796a39cd7b34239'
 }
```

Here’s the first 4 blocks in our chain.

Notice how each block's prevHash value matches the thisHash value of the block preceding it. This links all the blocks together in something similar to a linked-list, and ensures that tampering with the blockchain is extremely difficult.

**Finished!**

Using a few dozen lines of code, we've created our own blockchain!