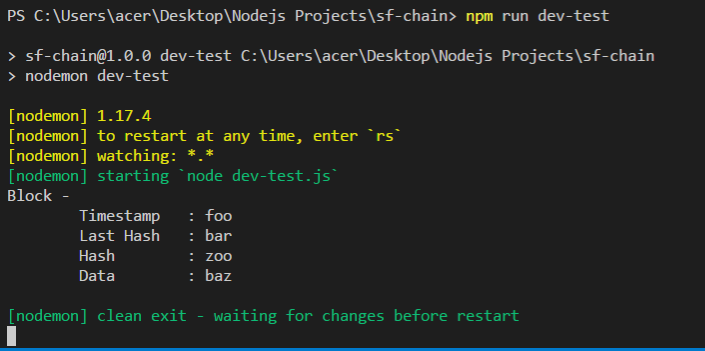
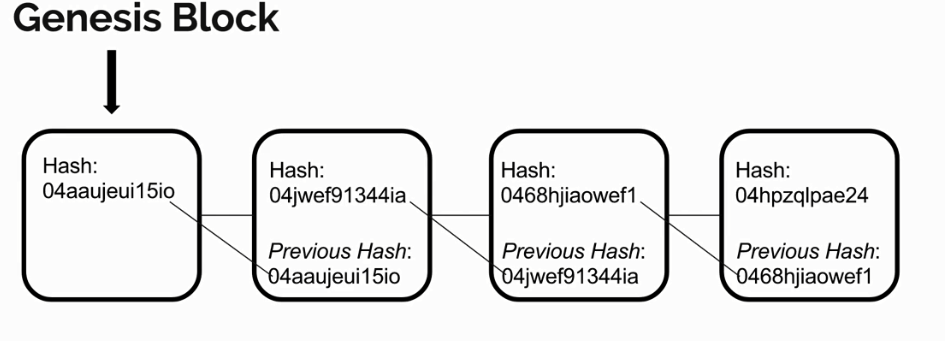
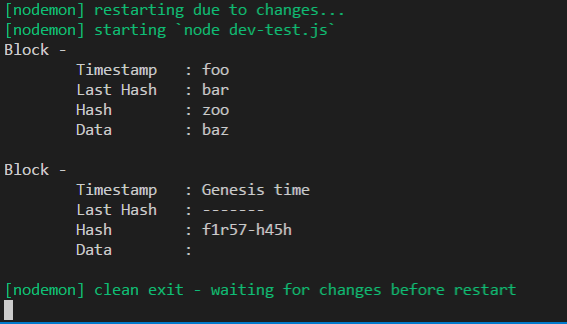
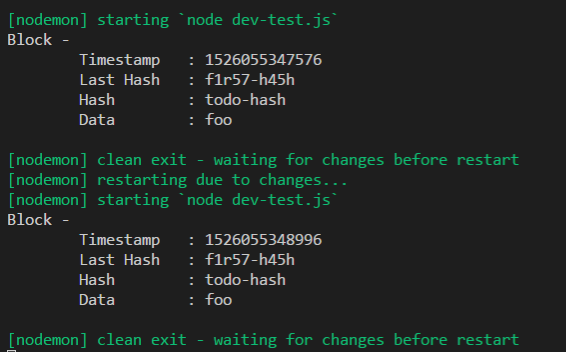
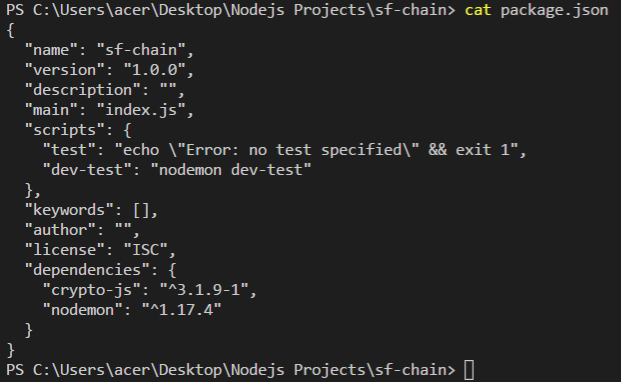
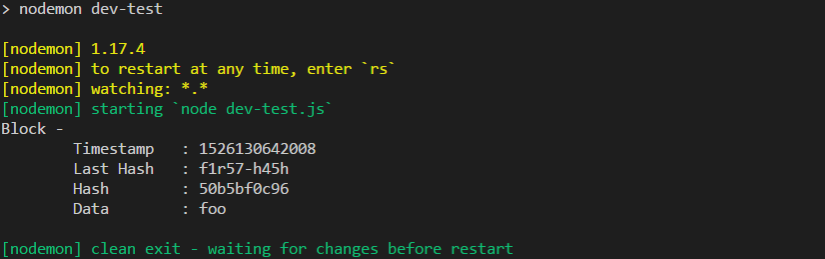
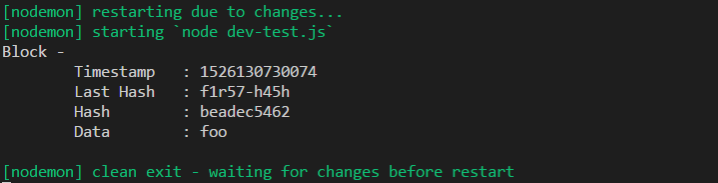
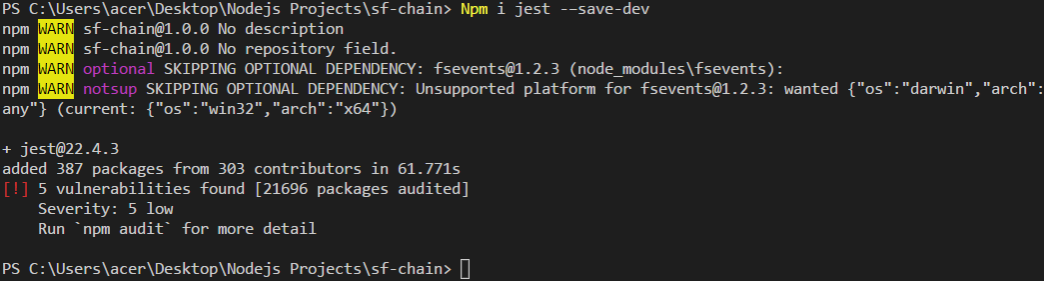
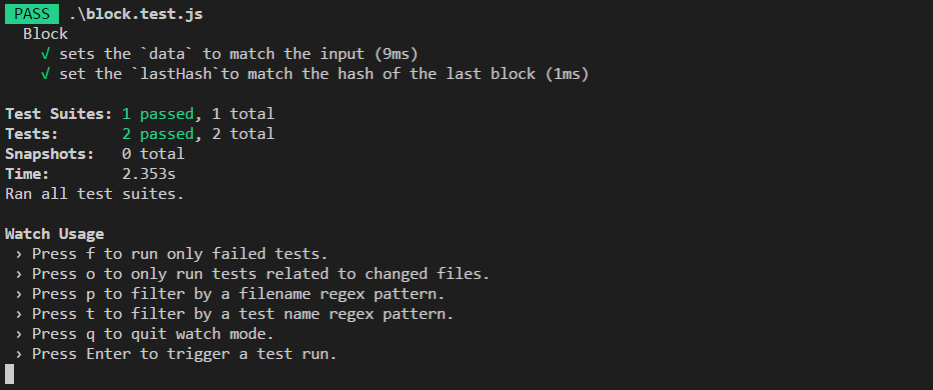
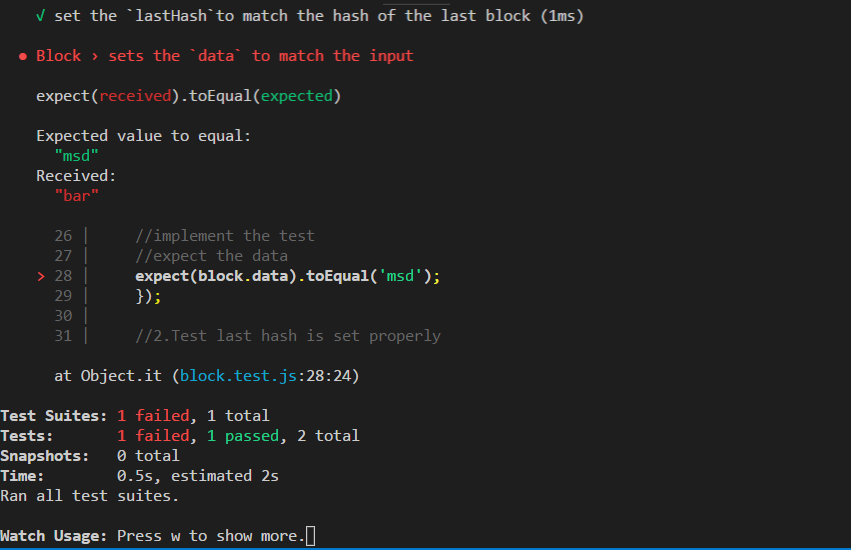
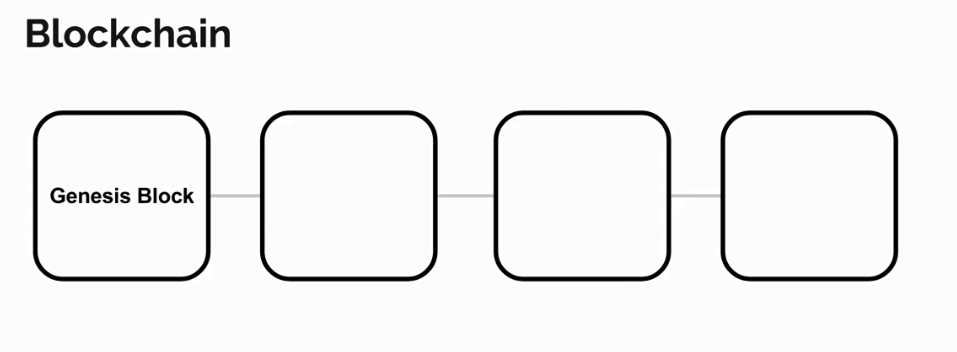
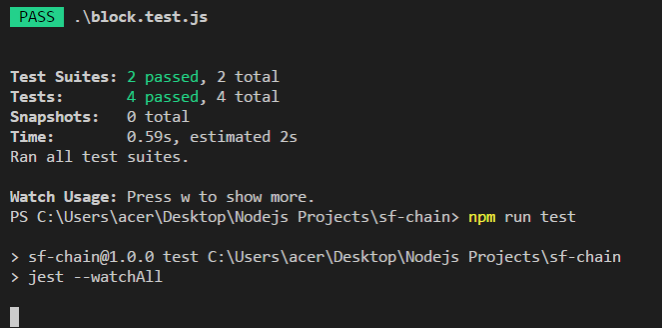
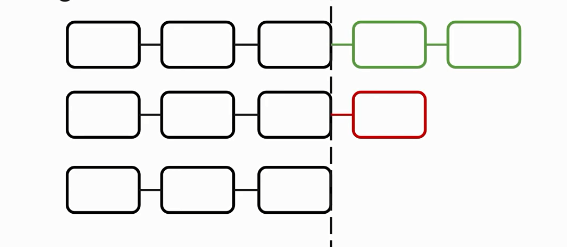
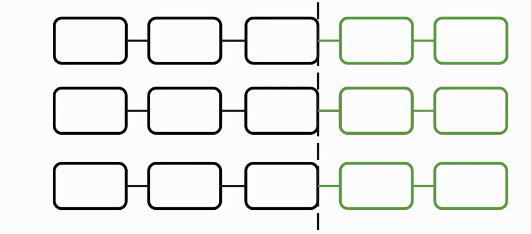
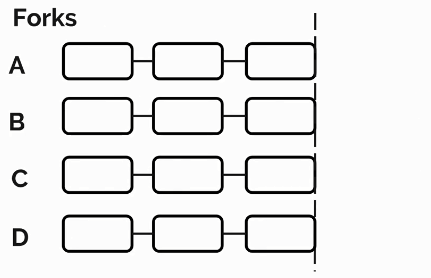
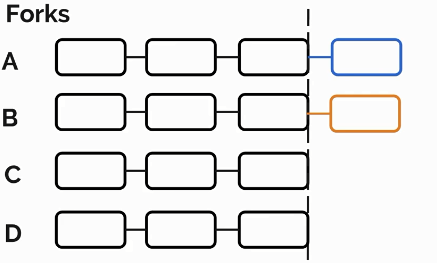
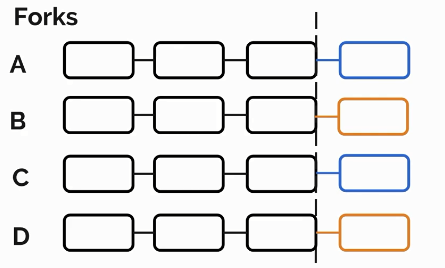
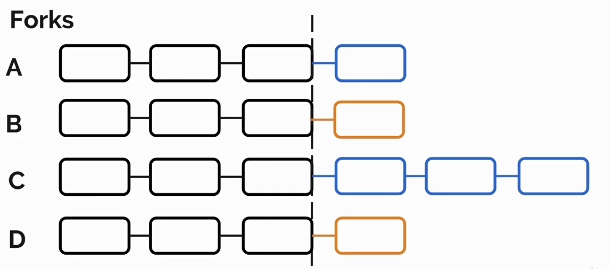
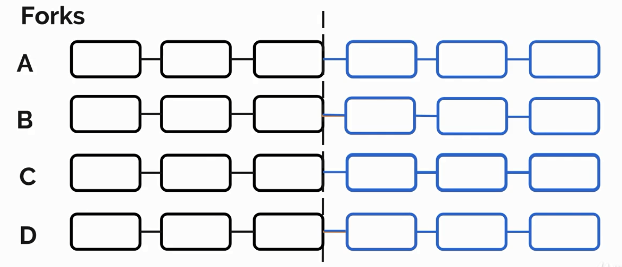
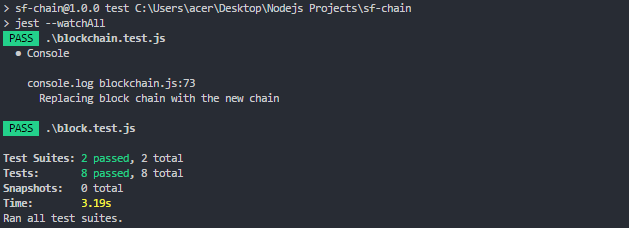
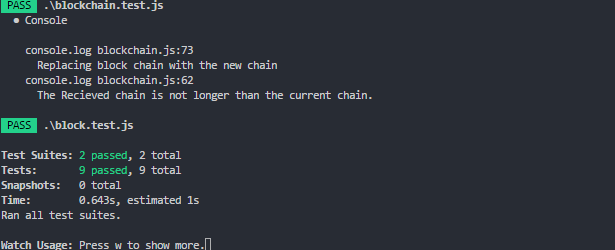
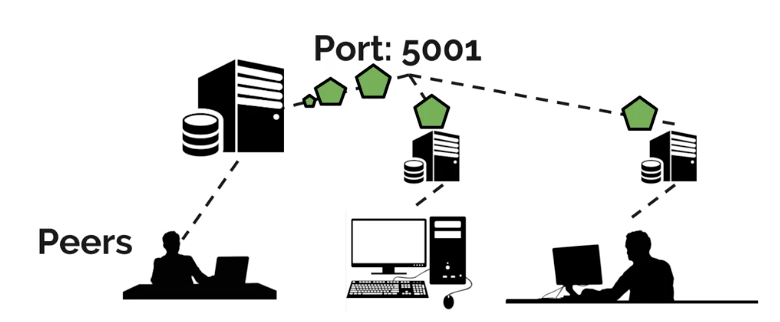
* Created a project and enter the project
* md sf-chain
* cd sf-chain
* Add a package.json and initialize
* Npm init –y
* Installing the nodeman for the project to see the project while developing
* nodemon –save-dev
* **Block**
* **Most fundament to power cryptocurrency**
* **To build block chain we need the blocks**
* **Block consist of 4 fundamental data**
* **Timestamp in milliseconds**
* **lastHash –the hash of the block before it**
* **(unique value for a block that is generated based on a combination all of its unique data)**
* **hash-based on it own data (Generated based on its own data)**
* **The data to store (Transction or any other data)**
* **We will be using object oriented style**
* **Creating the first block**
* **Block.js**
* class Block{
* //we call attributes that a block need
* constructor(timestamp,lastHash,hash,data){
* //input given to this constructor bind to the class
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* //will be using for debugging (in OOP)
* //for last hash 10 digit is enough
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* }
* //make sure the file is shared among others
* module.exports=Block;
* **Dev-test.js for the development purpose**
* //explore the block class
* const Block = require('./block')
* const block = new Block('foo','bar','zoo','baz');
* console.log(block.toString());
* **After typing the code we go to the package.json “dev-test”: “nodemon dev-test ” tell to run the dev test**
* {
* "name": "sf-chain",
* "version": "1.0.0",
* "description": "",
* "main": "index.js",
* "scripts": {
* "test": "echo \"Error: no test specified\" && exit 1",
* "dev-test": "nodemon dev-test"
* },
* "keywords": [],
* "author": "",
* "license": "ISC",
* "dependencies": {
* "nodemon": "^1.17.4"
* }
* }
* **Now in terminal run**
* **Npm run dev-test**
* 
* **Completed the block itself**
* **Genesis Block**
* **We defined blockchain as a chain of blocks**
* **Where each block generate a hash value that is calculated from the block came before.**
* **This makes the question for the first block that where the data come from if there is no block before.**
* **Block chains has the concept of the Genesis block**
* 
* **Genesis block surve the purpose of being the block chain. It is hard coded block with dummy values from the timestamp, last hash, hash and data value. It is a way for the block chain to get started. That way when some add a block to the chain that first block set it last hash to hash of the genesis block. Let’s created the genesis block in our project.**
* **We are in block.js we will add a new function called genesis block**
* **By giving the identifier of static we can call the function without creating its instance**
* class Block{
* //we call attributes that a block need
* constructor(timestamp,lastHash,hash,data){
* //input given to this constructor bind to the class
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* //will be using for debugging (in OOP)
* //for last hash 10 digit is enough
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* static genesis(){
* //this- return itself
* //last hash dmmy value, a dummy value, empty array
* return new this('Genesis time','-------','f1r57-h45h',[])
* }
* }
* //make sure the file is shared among others
* module.exports=Block;
* **in dev test**
* //explore the block class
* const Block = require('./block')
* const block = new Block('foo','bar','zoo','baz');
* console.log(block.toString());
* console.log(Block.genesis().toString());
* **See the result**
* 
* **We constructed the gensis block.**
* **Now lets generate the other blocks. We will capture this ability with mine block function.**
* **Mine block function will also be static, we need a new block instance in order to use this function. The argument for the functions are the last block and the data we want to store in the block.**
* class Block{
* //we call attributes that a block need
* constructor(timestamp,lastHash,hash,data){
* //input given to this constructor bind to the class
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* //will be using for debugging (in OOP)
* //for last hash 10 digit is enough
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* static genesis(){
* //this- return itself
* //last hash dmmy value, a dummy value, empty array
* return new this('Genesis time','-------','f1r57-h45h',[])
* }
* static mineBlock(lastBlock, data){
* //generate a timestamp
* const timestamp= Date.now();
* //local hash constant assign to the hash of the last block
* const lastHash= lastBlock.hash;
* //local hash constant (for now we dont have a hash function)
* const hash= 'todo-hash';
* //return the new instance of the block (it is same as genesis)
* return new this (timestamp,lastHash, hash, data);
* }
* }
* //make sure the file is shared among others
* module.exports=Block;
* **and the dev-test**
* //explore the block class
* const Block = require('./block')
* //Used for the test before the mine block function
* /\*const block = new Block('foo','bar','zoo','baz');
* console.log(block.toString());
* console.log(Block.genesis().toString()); \*/
* //Minning the new block / we need to give the last block as referance
* //It is the first block so (give the ginese block)
* const fooBlock= Block.mineBlock(Block.genesis(), 'foo'); //for the foo block
* console.log(fooBlock.toString());
* **for getting the first block we assign the costant foo block which calls the static gensis function. It loads the mineBlock function and use foo as the store data**
* 
* **Show that we have a successful mine block function. It has a unique timestamp which is a lot of mili-seconds (Note we used the substring method ) almost 50 years ago**
* **Lets generate the unique hash value with new function that generate the hash.**
* **SHA256 Hash Function**
* The hash is generated from the timestamp, last Hash and the store data.
* For the unique hash value we will use the unique algorithm called SHA-256, which stands for secure H Algorithm-256. The 256 part represents the fix size of 256-bit hash which is 32 byte (32 characters) will be generated to represent the Hash, so this hashing algorithms has nice properties:
* Even if one character changes in the original data SHA-256 algorithm will produce an entire unique string of characters for a different hash but it will always produce the same hash with given data.
* One-way hash function, it is impossible to decrypt that hash and get the data back. It only works one way from the direction of the data to the hash. This is useful when we come to the
* Useful for block validation, we can immediately know if the data has temper with, if it generated data is different from one that we create given the data that is in the block.
* Lets add this to the project, make sure the sf-chain is open
* Adding new model name **crypto-js** contain an SHA-256 function that we can use,
* Npm i crypto-js –save
* To check the package is package.json we write
* Cat package.json
* 
* We have the dependency crypt-js installed now
* At the top of the block.js file access to the SHA-256 function
* //require from the crypto-js
* //Generate a unique hash based on the provide data
* const SHA256=require('crypto-js/sha256')
* class Block{
* //we call attributes that a block need
* constructor(timestamp,lastHash,hash,data){
* //input given to this constructor bind to the class
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* //will be using for debugging (in OOP)
* //for last hash 10 digit is enough
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* static genesis(){
* //this- return itself
* //last hash dmmy value, a dummy value, empty array
* return new this('Genesis time','-------','f1r57-h45h',[])
* }
* static mineBlock(lastBlock, data){
* //generate a timestamp
* const timestamp= Date.now();
* //local hash constant assign to the hash of the last block
* const lastHash= lastBlock.hash;
* //local hash constant (for now we dont have a hash function)
* const hash= Block.hash(timestamp,lastHash,data);
* //return the new instance of the block (it is same as genesis)
* return new this (timestamp,lastHash, hash, data);
* }
* //static hash function: represent the unique data that we want to generate the hash for
* //argument the time stamp of the block, last hash and the data the block is storing
* static hash(timestamp,lastHash, data){
* //combing as one string using ES6 string
* return SHA256(`${timestamp} ${lastHash}${data}`).toString();
* //replace the to do with hash
* }
* }
* //file is shared among others
* module.exports=Block;
* created the hash function and equalize hash constant in the mine block to the hash function that uses the SHA-256 algorithms from the crypto-js module.
* Run the dev-test and check the result
* Npm run dev-test
* 
* Hash will be different each time because the time stamp will be different each time
* 
* As we call block.js within the test-dev file we now have a hash value based on its timestamp and the hash of the last block that we provided as well as the data that we want to store. The Generation of hashes based on the last block is what links blocks togather ultimately making the entire block chain.
* We test this project continuously via console log but for the good result we need to develop a robust test environment. In that we will test this class and any other class we add to this project. These test will serve as documentations for how to use these classes and these objects.
* For test we need to install the jest module
* Npm i jest –save-dev
* With jest we can execute the testing files even the javascript projects. To find the test file we will look for the tes.js extension. To test our block class we will create the Block.test.js file
* 
* In block.js we add the SHA-256 alogrithms
* //require from the crypto-js
* //Generate a unique hash based on the provide data
* const SHA256=require('crypto-js/sha256')
* class Block{
* //we call attributes that a block need
* constructor(timestamp,lastHash,hash,data){
* //input given to this constructor bind to the class
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* //will be using for debugging (in OOP)
* //for last hash 10 digit is enough
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* static genesis(){
* //this- return itself
* //last hash dmmy value, a dummy value, empty array
* return new this('Genesis time','-------','f1r57-h45h',[])
* }
* static mineBlock(lastBlock, data){
* //generate a timestamp
* const timestamp= Date.now();
* //local hash constant assign to the hash of the last block
* const lastHash= lastBlock.hash;
* //local hash constant (for now we dont have a hash function)
* const hash= Block.hash(timestamp,lastHash,data);
* //return the new instance of the block (it is same as genesis)
* return new this (timestamp,lastHash, hash, data);
* }
* //static hash function: represent the unique data that we want to generate the hash for
* //Providing argument the time stamp of the block, last hash and the data the block is storing
* static hash(timestamp,lastHash, data){
* //combing as one string using ES6 string
* return SHA256(`${timestamp} ${lastHash}${data}`).toString();
* //replace the to do with hash
* }
* }
* //file is shared among others
* module.exports=Block;
* Creating the file Block.test.js
* //Access to the block module
* const Block= require('./block');
* //providing the test the block and the error call back function
* describe('Block', ()=>{
* //Declaring variables
* let data, lastBlock, block;
* beforeEach(()=>{
* //assign the data dummy value
* data='bar';
* //default last block
* lastBlock= Block.genesis();
* //mine the block
* block= Block.mineBlock(lastBlock, data);
* });
* //for unit test using just function it
* //it (description about test executed, call back error function codes to execute the test)
* //1.Test the block to set the data to the matched input
* it('sets the `data` to match the input',()=>{
* //implement the test
* expect(block.data).toEqual(data);
* });
* //2.Test last hash is set properly
* it('set the `lastHash`to match the hash of the last block',()=> {
* expect (block.lastHash).toEqual(lastBlock.hash)
* });
* });
* //beforeach function: allow the same code for each of the following unit test
* Go to package.json and change the test to the jest
* "test": "jest --watchAll",
* Similar to the nodeman it sets up a server that listen the changes and re run the entire sweep when ever it detects a new file with the change is placed
* Head to the terminal
* Npm run test
* 
* If you want to check this result go the block.test.js and change the test equalize value you can see
* 
* After you change the values back
* Block Chain
* Block chain is based on a few concept, chain is an array of block, and First block is the gensis block. It has the ability to add a block to the chain and added to the end of its block. And new block is created based on the last block. By doing this it creates link across the chain.
* 
* To start add a new file to the root of the project called blockchain.js. This will hold the block chain class
* //Access to the required block module
* const Block = require('./block');
* //creating the block chain
* class Blockchain{
* constructor(){
* //chain of array (gensis block)
* this.chain=[Block.genesis()];
* }
* //a function for this class
* //parameter:Data we want to store
* addBlock(data){
* //we need last block that will take the value from the last index
* //last value of this chain array
* const lastBlock=this.chain[this.chain.length-1];
* //Generate new block
* const block= Block.mineBlock(lastBlock, data);
* //adding the new produced block to the chain array
* this.chain.push(block)
* //result of the function
* return block;
* }
* }
* //sharing this module to be accessed by other files
* module.exports= Blockchain;
* Create the blockchain.test.js file in the root folder
* //Access to the required blockchain module
* const Blockchain = require('./blockchain');
* //require the block model that has gunesis function
* const Block= require('./block')
* //providing the test the blockchain and the error call back function
* describe('Blockchain', () => {
* //Declaring variables
* let bc;
* //beforeEach runs before each unit test
* beforeEach(() => {
* //new-keyword: refresh instance of this blockchain class
* bc= new Blockchain();
* });
* //it (description about test executed, call back error function codes to execute the test)
* //1.Test the blockchain to start with genesis block
* it('starts with genesis block', () => {
* //the first value in the chain equal to genesis
* expect(bc.chain[0]).toEqual(Block.genesis());
* });
* //2.Test adds a new block
* it('adds a new block', () => {
* const data = 'foo';
* bc.addBlock(data);
* //Look at the last block the chain that is same as dummy data
* expect(bc.chain[bc.chain.length-1].data).toEqual(data)
* });
* });
* //beforeach function: allow the same code for each of the following unit test
* And run the test from terminal
* 
* Multiple chain Validation
* Extend to multiple contributors
* Each miner will have their own version of the chain
* Say when miners add a new block to the chain, they will have to submit that new block for the other chain in the system to accept that in order to update the overall system so that way everyone gets an updated copy with that new block loaded in. that way all the changes are consistent. How ever for other miners to accept the changes there need to be some validation to make sure that new block is valid and that they should actually accept.
* Longer Chains
* 
* Suppose we have three block chains one miner add two and an other miner add one. Accepting longer chains come in that block chain system can ensure that the agreed upon the change for everyone is always the one that has the most data with in it. 
* This also produces a situation which result a problem of forks in the chain
* 
* Miner a add a block and miner b add a block so 
* they create a fork in the system based on the last block
* 
* So now half has a chain submitted by miner and half have the chain submitted by miner B.
* Says a miner exten upon a miner change. This chain now longer than everyone chain in the system. Therefoe everyone have to accept the Long chain.
* 
* This Longer chain has miner a blockfrom the fork with in it. Now fork is resolved by accepting the orginal block submitted by miner A. this not means block chain which had the block from miner B will lose that original block data. The block can be added to the newly received chain.
* 
* Hash Validation:
* This validation is check the has value produced by each chain of the block, so every block chain has the access to hash function which generate the hash base of the block data. When a block chain receive a change it can ensure that this hash is regenerated by regenerating this hash itself. If the hashes don’t match data might probably temped with therefore the block chain will decide not accept the new change.
* Lets add this validation functionality to the project.
* Adding a fuction that will check the Chain that does the job of validating the chain . If the chain is invalid the function will return false, otherwise the chain is okay the function returns True.
* In block chain js we add is valid method
* *//Access to the required block module*
* const Block = require('./block');
* *//creating the block chain*
* class Blockchain{
* constructor(){
* *//chain of array (gensis block)*
* this.chain=[Block.genesis()];
* }
* *//a function for this class*
* *//parameter:Data we want to store*
* addBlock(data){
* *//we need last block that will take the value from the last index*
* *//last value of this chain array*
* const lastBlock=this.chain[this.chain.length-1];
* *//Generate new block*
* const block= Block.mineBlock(lastBlock, data);
* *//adding the new produced block to the chain array*
* this.chain.push(block)
* *//result of the function*
* return block;
* }
* *//function that will check the validityi of incoming chain*
* *//Parameters (incoming chain)*
* isValidChanin(chain){
* *//first element of incoming chain matches the genesis block*
* *//Stringyify the element to compare*
* if (JSON.stringify(chain[0])!== JSON.stringify(Block.genesis())) return false;
* *//validation on each block after teh genesis block in the incoming chain*
* for (let i = 0; i < chain.length; i++) {
* *//current block*
* const block=chain[i];
* *//the last block*
* const lastBlock=chain[i-1];
* *//the current block last hash must match the hash of the last block*
* *//if the block hash not equal to the generated hash (changed by any reason)*
* if (block.lashHash !== lastBlock.hash || block.lashHash!==Block.blockHash(block)) {
* return false;
* }
* }
* return true;
* }
* }
* *//sharing this module to be accessed by other files*
* module.exports= Blockchain;
* in block block js we implement the method to generate the hashes
* *//for generating hashes*
* static blockHash(block){
* *//all values to generate hash*
* const {timestamp, lastHash, data}= block;
* return Block.hash(timestamp,lastHash,data);
* }
* Block.js will look like
* *//require from the crypto-js*
* *//Generate a unique hash based on the provide data*
* const SHA256=require('crypto-js/sha256')
* class Block{
* *//we call attributes that a block need*
* constructor(timestamp,lastHash,hash,data){
* *//input given to this constructor bind to the class*
* this.timestamp=timestamp;
* this.lastHash=lastHash;
* this.hash=hash;
* this.data=data;
* }
* *//will be using for debugging (in OOP)*
* *//for last hash 10 digit is enough*
* toString(){
* return `Block -
* Timestamp : ${this.timestamp}
* Last Hash : ${this.lastHash.substring(0,10)}
* Hash : ${this.hash.substring(0, 10)}
* Data : ${this.data}
* `
* }
* static genesis(){
* *//this- return itself*
* *//last hash dmmy value, a dummy value, empty array*
* return new this('Genesis time','-------','f1r57-h45h',[])
* }
* static mineBlock(lastBlock, data){
* *//generate a timestamp*
* const timestamp= Date.now();
* *//local hash constant assign to the hash of the last block*
* const lastHash= lastBlock.hash;
* *//local hash constant (for now we dont have a hash function)*
* const hash= Block.hash(timestamp,lastHash,data);
* *//return the new instance of the block (it is same as genesis)*
* return new this (timestamp,lastHash, hash, data);
* }
* *//static hash function: represent the unique data that we want to generate the hash for*
* *//Providing argument the time stamp of the block, last hash and the data the block is storing*
* static hash(timestamp,lastHash, data){
* *//combing as one string using ES6 string*
* return SHA256(`${timestamp} ${lastHash}${data}`).toString();
* *//replace the to do with hash*
* }
* *//for generating hashes*
* static blockHash(block){
* *//all values to generate hash*
* const {timestamp, lastHash, data} = block;
* return Block.hash(timestamp,lastHash, data);
* }
* }
* *//file is shared among others*
* module.exports=Block;
* in the Blockchain.Test.js we implement this third unity test
* *//3.Test validate a valid chain*
* it('validates a valid chain', () => {
* *//declar variable BC2, assign new new block chain instance*
* bc2.addBlock('foo');
* expect(bc.isValidChain(bc2.chain)).toBe(true);
* *//run the test script*
* });
* So blockchain.test.js will look like*//Access to the required blockchain module*
* const Blockchain = require('./blockchain');
* *//require the block model that has gunesis function*
* const Block= require('./block')
* *//providing the test the blockchain and the error call back function*
* describe('Blockchain', () => {
* *//Declaring variables*
* let bc, bc2;
* *//beforeEach runs before each unit test*
* beforeEach(() => {
* *//new-keyword: refresh instance of this blockchain class*
* bc= new Blockchain();
* bc2= new Blockchain();
* });
* *//it (description about test executed, call back error function codes to execute the test)*
* *//1.Test the blockchain to start with genesis block*
* it('starts with genesis block', () => {
* *//the first value in the chain equal to genesis*
* expect(bc.chain[0]).toEqual(Block.genesis());
* });
* *//2.Test adds a new block*
* it('adds a new block', () => {
* const data = 'foo';
* bc.addBlock(data);
* *//Look at the last block the chain that is same as dummy data*
* expect(bc.chain[bc.chain.length-1].data).toEqual(data)
* });
* *//3.Test validate a valid chain*
* it('validates a valid chain', () => {
* bc2.addBlock('foo');
* expect(bc.isValidChain(bc2.chain)).toBe(true);
* });
* *//4.Test invalidates with a corrupt genesis block*
* it('invalidates a chain with a corrupt genesis block', () => {
* *//incase bad data is in genesis block*
* bc2.chain[0].data = 'Bad data';
* expect(bc.isValidChain(bc2.chain)).toBe(false);
* });
* *//5.Test invalidates a corrupt chain \*(Data is tempered)*
* it('invalidates a corrupt chain', () => {
* *//incase add block is changes then*
* bc2.addBlock('foo');
* bc2.chain[1].data = 'Not foo';
* expect(bc.isValidChain(bc2.chain)).toBe(false);
* });
* });
* *//beforeach function: allow the same code for each of the following unit test*
* So run the unit tests
* npm run test
* \*Work with the jest, and search for all the .Test Files
* Our goal is to add multiple collaborators to the block.
* We added the validation method for the chain to validate the chain now we will add the actual function that replaces the chain with incoming block chain if it is valid. So we will the function replace change in the blockchain.js
* replaceChain(newChain){
* *//make sure it is a valid chain (Use valid chain function)*
* if (newChain.length<=this.chain.length){
* console.log('The Recieved chain is not longer than the current chain.')
* *//escape from the function*
* return;
* }
* For example the fork problem if two minner submit one block at the same time the longer chain is acceptable for both of them. And resolve the fork issue.
* We can add additional check
* replaceChain(newChain){
* *//make sure it is a longer chain than the current chain*
* if (newChain.length<=this.chain.length){
* console.log('The Recieved chain is not longer than the current chain.')
* *//escape from the function*
* return;
* }
* *//check the new change is valid*
* else if(!this.isValidChain(newChain)){
* console.log('The recieved chain is not valid.');
* *//escape from the function*
* return
* }
* *//after passing the following controlls*
* console.log('Replacing block chain with the new chain');
* this.chain=newChain;
* }
* Now blockchain.js will look like
* *//Access to the required block module*
* const Block = require('./block');
* *//creating the block chain*
* class Blockchain{
* constructor(){
* *//chain of array (gensis block)*
* this.chain=[Block.genesis()];
* }
* *//a function for this class*
* *//parameter:Data we want to store*
* addBlock(data){
* *//we need last block that will take the value from the last index*
* *//last value of this chain array*
* const lastBlock=this.chain[this.chain.length-1];
* *//Generate new block*
* const block= Block.mineBlock(lastBlock, data);
* *//adding the new produced block to the chain array*
* this.chain.push(block)
* *//result of the function*
* return block;
* }
* *//function that will check the validityi of incoming chain*
* *//Parameters (incoming chain)*
* isValidChain(chain){
* *//first element of incoming chain matches the genesis block*
* *//Stringyify the element to compare*
* if (JSON.stringify(chain[0]) !== JSON.stringify(Block.genesis())) return false;
* *//validation on each block after teh genesis block in the incoming chain*
* for (let i = 1; i < chain.length; i++) {
* *//current block*
* const block = chain[i];
* *//the last block*
* const lastBlock = chain[i - 1];
* *//the current block last hash must match the hash of the last block*
* *//if the block hash not equal to the generated hash (changed by any reason)*
* if (block.lastHash !== lastBlock.hash ||
* block.hash !== Block.blockHash(block)) {
* return false;
* }
* }
* return true;
* }
* replaceChain(newChain){
* *//make sure it is a longer chain than the current chain*
* if (newChain.length<=this.chain.length){
* console.log('The Recieved chain is not longer than the current chain.')
* *//escape from the function*
* return;
* }
* *//check the new change is valid*
* else if(!this.isValidChain(newChain)){
* console.log('The recieved chain is not valid.');
* *//escape from the function*
* return
* }
* *//after passing the following controlls*
* console.log('Replacing block chain with the new chain');
* this.chain=newChain;
* }
* }
* *//sharing this module to be accessed by other files*
* module.exports= Blockchain;
* in blockchain.Test.js we will write a unity test
* *//Test for the replaceChain function in the Block.js*
* *//1.Test: Check if the chain is replaced with a valid chain*
* it('Testing: Replaces the chain with a valid chain', () => {
* *//add a new block*
* bc2.addBlock('goodness');
* *// pass the first change to the blockchain it should valid*
* bc.replaceChain(bc2.chain);
* *//check the change of the first instance should equal to the change of second instance*
* expect(bc.chain).toEqual(bc2.chain);
* });
* And run the test
* 
* We add the second test too
* *//Test for the replaceChain function in the Block.js*
* *//1.Test: Check if the chain is replaced with a valid chain*
* it('Testing: Replaces the chain with a valid chain', () => {
* *//add a new block*
* bc2.addBlock('goodness');
* *// pass the first change to the blockchain it should valid*
* bc.replaceChain(bc2.chain);
* *//check the change of the first instance should equal to the change of second instance*
* expect(bc.chain).toEqual(bc2.chain);
* });
* *//2.Test: Check if the chain is replaced with a valid chain*
* it('It does not replace the chain with one of less than or equal to length', () => {
* bc.addBlock('wee');
* *// pass the first change to the blockchain it should valid*
* bc.replaceChain(bc2.chain);
* *//check the change of the first instance should not equal with the change of second instance*
* expect(bc.chain).not.toEqual(bc2.chain);
* });
* And now the Block.Test.js will look like
* *//Access to the required blockchain module*
* const Blockchain = require('./blockchain');
* *//require the block model that has gunesis function*
* const Block= require('./block')
* *//providing the test the blockchain and the error call back function*
* describe('Blockchain', () => {
* *//Declaring variables*
* let bc, bc2;
* *//beforeEach runs before each unit test*
* beforeEach(() => {
* *//new-keyword: refresh instance of this blockchain class*
* bc= new Blockchain();
* bc2= new Blockchain();
* });
* *//it (description about test executed, call back error function codes to execute the test)*
* *//1.Test the blockchain to start with genesis block*
* it('starts with genesis block', () => {
* *//the first value in the chain equal to genesis*
* expect(bc.chain[0]).toEqual(Block.genesis());
* });
* *//2.Test adds a new block*
* it('adds a new block', () => {
* const data = 'foo';
* bc.addBlock(data);
* *//Look at the last block the chain that is same as dummy data*
* expect(bc.chain[bc.chain.length-1].data).toEqual(data)
* });
* *//3.Test validate a valid chain*
* it('validates a valid chain', () => {
* bc2.addBlock('foo');
* expect(bc.isValidChain(bc2.chain)).toBe(true);
* });
* *//4.Test invalidates with a corrupt genesis block*
* it('invalidates a chain with a corrupt genesis block', () => {
* *//incase bad data is in genesis block*
* bc2.chain[0].data = 'Bad data';
* expect(bc.isValidChain(bc2.chain)).toBe(false);
* });
* *//5.Test invalidates a corrupt chain \*(Data is tempered)*
* it('invalidates a corrupt chain', () => {
* *//incase add block is changes then*
* bc2.addBlock('foo');
* bc2.chain[1].data = 'Not foo';
* expect(bc.isValidChain(bc2.chain)).toBe(false);
* });
* *//Test for the replaceChain function in the Block.js*
* *//1.Test: Check if the chain is replaced with a valid chain*
* it('Testing: Replaces the chain with a valid chain', () => {
* *//add a new block*
* bc2.addBlock('goodness');
* *// pass the first change to the blockchain it should valid*
* bc.replaceChain(bc2.chain);
* *//check the change of the first instance should equal to the change of second instance*
* expect(bc.chain).toEqual(bc2.chain);
* });
* *//2.Test: Check if the chain is replaced with a valid chain*
* it('It does not replace the chain with one of less than or equal to length', () => {
* bc.addBlock('wee');
* *// pass the first change to the blockchain it should valid*
* bc.replaceChain(bc2.chain);
* *//check the change of the first instance should not equal with the change of second instance*
* expect(bc.chain).not.toEqual(bc2.chain);
* });
* });
* *//beforeach function: allow the same code for each of the following unit test*
* We run the test and receive the resuls
* 
* **Develop the Blockchain Application**
* Organize the project to keep things clean. Create a folder at the root called block chain and move the files into it.
* Organized into the folders
* Blockchain is changed to index
* Help user to interact with the application
* Through API Users will be able to view the blocks currently held in of our blockchain instances.
* They will be able to add mining through minning end point which will add new blocks to the chain .
* We will use the express module.
* Npm I express –save
* const express= require ('express');
* const Blockchain=require('../blockchain');
* *//default page index.js*
* *//find the port application listen for request*
* *//const HTTP\_PORT=3001; //acces through localhost domain*
* const HTTP\_PORT= process.env.HTTP\_PORT || 3001; *//Multiple run time at same time*
* *//Create an express application*
* const app=express();
* const bc= new Blockchain();
* *//first end point for API*
* *//interact to user*
* app.get('/blocks', (req,res)=>{
* res.json(bc.chain);
* });
* app.listen(HTTP\_PORT, ()=>console.log(`Listening on port ${HTTP\_PORT}`));
* in package.json
* {
* "name": "sf-chain",
* "version": "1.0.0",
* "description": "",
* "main": "index.js",
* "scripts": {
* "test": "jest --watchAll",
* "dev-test": "nodemon dev-test",
* "start":"node ./app",
* "dev":"nodemon ./app"
* },
* "keywords": [],
* "author": "",
* "license": "ISC",
* "dependencies": {
* "crypto-js": "^3.1.9-1",
* "jest": "^22.4.3",
* "nodemon": "^1.17.4"
* }
* }
* Npm run dev //starting the project
* Port listen to 2001 and Used postman localhost:2001/blocks
* Npm i body-praser –save
* Peer-to-peer Server
* We want to set up peer-to-peer servers, for the multi collaborator in the block chain. Every individual running the block chain application will be considered peer. The first pair starts the application and opens the peer-to-peer server.
* We created the UI for the Web API so that we can add a block and remove a block. We can add a block to the chain using the postman from the local host.
* 
* We will use a technology called web socket for this peer to peer service, which allow us to set up real time virtual connection between multiple nodes running in node.js application. It will open a port where it will listen for the web socket connection as other nodes fire up the application they will set up their own web socket server too. But more importantly they now connect to the original server through that open socket port. Like wise the original server detects the new instances of the application have attempted to connect to its web socket server. It makes a connection over the web socket port to the instances.
* Overall through this connection the peers has the ability to broadcast data through stringify messages to all the connected pairs.



* As each new peer connect to our block chain application we need to ensure that the peer has the updated block chain.
* As new blocks are added by one peer they will have the ability to broadcast the change in the data of the block to all the connected pairs over the web socket.
* There are two possibilities on how the server will behave;
* 1. The first application to stat the peer-to-peer server, then it will wait for the connection as its peer application start up
* 2. A later web socket server, this node will connect to the original peer-to-peer server.
* In our implementation we won’t have separate classes for both. One peer-to-peer server itself will have both the capabilities for part 1 and part 2. However it will know whether or not to open the server or to connect to pairs immediately based on us providing the environment variable we specify for what it has Building the application adding multiple collaborators to the block chain application

We want to add multiple collaborators to our block chain server; we want to have at least two servers. Whenever one block is add to a running block chain, all the running block chain receive that change and update the changes accordingly.

We need access to a new module the web socket module.

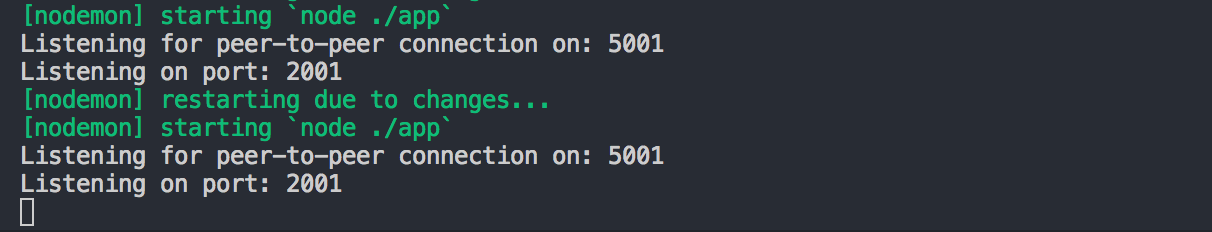
npm i ws --save

* Create a file called p2p-server.js (peer-to-peer) to write the `P2pServer` class. Right now the `P2pServer` class will open a websocket server, waiting for connections.
* ```
* const Websocket = require('ws');
* const P2P\_PORT = process.env.P2P\_PORT || 5001;

//List of web socket adress

* const peers = process.env.PEERS ? process.env.PEERS.split(',') : [];
* class P2pServer {
* constructor(blockchain) {
* this.blockchain = blockchain;
* this.sockets = [];
* }
* listen() {
* const server = new Websocket.Server({ port: P2P\_PORT });
* server.on('connection', socket => this.connectSocket(socket));
* }
* connectSocket(socket) {
* this.sockets.push(socket);
* console.log('Socket connected');
* }
* }

module.exports = P2pServer;



**Connect Peers**

The same class that creates the original websocket server will be used to connect to existing servers.

* const Websocket = require('ws');
* const P2P\_PORT = process.env.P2P\_PORT || 5001;
* const peers = process.env.PEERS ? process.env.PEERS.split(','):[];
* class p2pServer{
* constructor(blockchain){
* this.blockchain = blockchain;
* this.sockets = [];
* }
* // For listening the
* listen(){
* // Server class
* const server = new Websocket.Server({port: P2P\_PORT});
* // Listen for the incoming messages
* // Fire when new socket connected call back function will do work
* server.on('connection', socket=>this.connectSocket(socket));
* this.connectToPeers();
* console.log(`Listening for peer-to-peer connections on: ${P2P\_PORT}`);
* }
* connectToPeers(){
* // For each pairs
* peers.forEach(peer => {
* // ws:localhost:501
* const socket = new Websocket(peer);
* // start server
* socket.on('open', () => this.connectSocket(socket))
* });
* }
* connectSocket(socket){
* this.sockets.push(socket);
* console.log('Socket connected');
* }
* }

module.exports= p2pServer;

The next step is to go to the index.js and import the P2pServer:

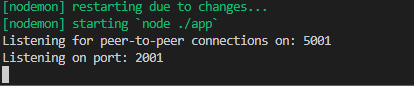
* const P2pServer = require('./p2p-server'), and we create an instance below the block chain:
* const app = express();
* const bc = new Blockchain();

const P2pServer = new P2pServer(bc)

* the p2pServer takes an argument which is a block chain. (bc is the local instance of the block chain). In the end we:
* // Start the web socket server

P2pServer.listen();

In terminal we will run npm run dev



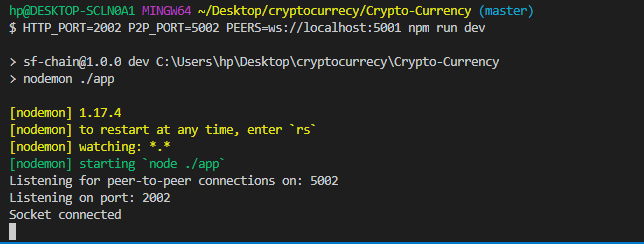
It will start a peer to peer server and a HTTP server for the API.

Since the default ports are taken we open another terminal and we will change the default port by changing the environment variable.

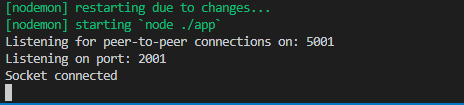
In another terminal:

HTTP\_PORT=3002 P2P\_PORT=5002 PEERS=ws://localhost:5001 npm run dev

now we can see that web socket is connected. And the server is started on the new port 2002 and p2p on 5002

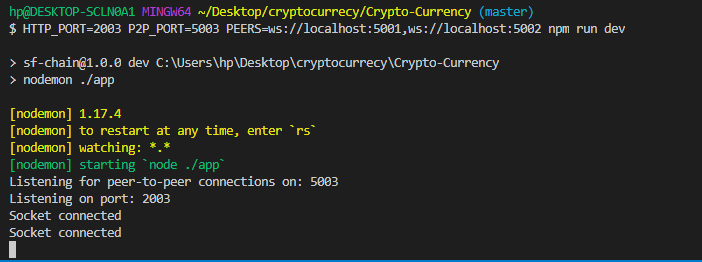


If you go to the first tab now you can see that socket is connected.



We can extend this to a third instance of the application by opening the new terminal tab type:

Now we have two peers first in local host port 5001 other on local host port 5002 and we have two sockets connected. (NOTE: other nodes needs to be running in order to connect them.)



We also added the previous one according to web socket address. Here we have two sockets connected. Now we share the block chains in order to have a truly decentralized block chain technology.

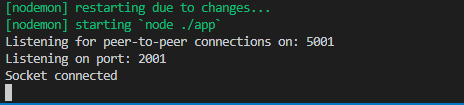
**Message Handler from Peers**

We have multiple sockets connected to our block chain. We have synchronized our sockets’ block chains. To allow our sockets to communicate and send the block chain specific data, we use the send method of the socket object.

The send method allows us to send an event to the relevant socket containing a stringfyed message. In this model we will be sending message event to sockets likewise we need to check our sockets are prepared to receive those message events, by creating an event listener at the bottom called message Handler. It will take a parameter called socket.

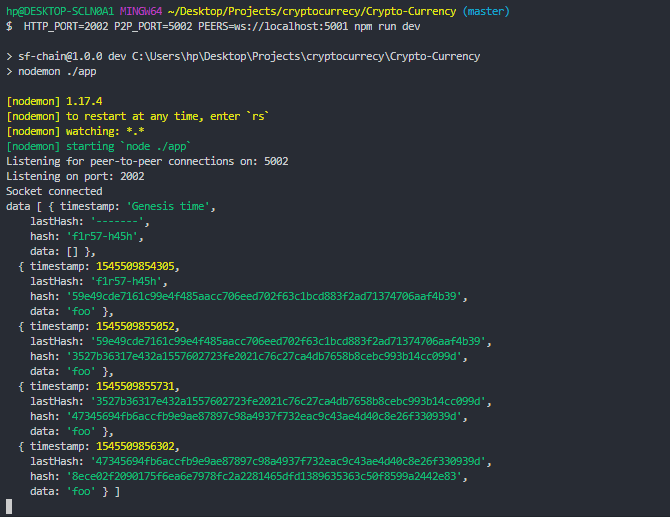
* In the `P2pServer` class:
* messageHandler(socket) {
* socket.on('message', message => {
* const data = JSON.pars(message);
* console.log('data', data);
* });
* });
* In `this.connectSocket`:
* connectSocket(socket) {
* this.sockets.push(socket);
* console.log('Socket connected');
* this.messageHandler(socket);
* // send the message event containing the block chain object to the sockets
* socket.send(JSON.stringify(this.blockchain.chain));
* }
* Each of our sockets is ready to receive a message event. Still we need to send the message event containing the block chain object to the sockets that we connect to.

In this part we will kill all the running terminal and mine some blocks. After mining the blcoks we will lunch the second instance of terminal



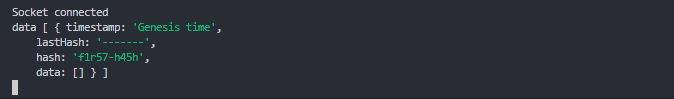
And then we run another terminal window and run

HTTP\_PORT=2002 P2P\_PORT=5002 PEERS=ws://localhost:5001 npm run dev



Then you can see the mined block.

Now if you check the first tab you can find an instance of the connected peer via web socket.



We need to make sure that these messages update our block chain. And they are all agreed on one, the second block chain also send its block instances to first and the first block chain is sending its block to the second.

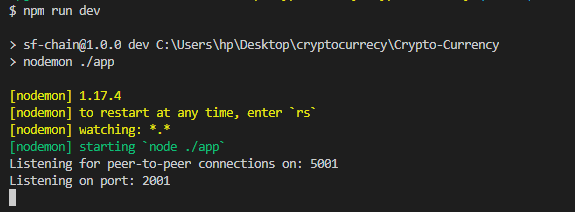
For test:

If new block chain appears the connect to that first app receive a message containing that entire block chain, after that second instance fire it will run the connected socket and should get the block chain of the first instance.

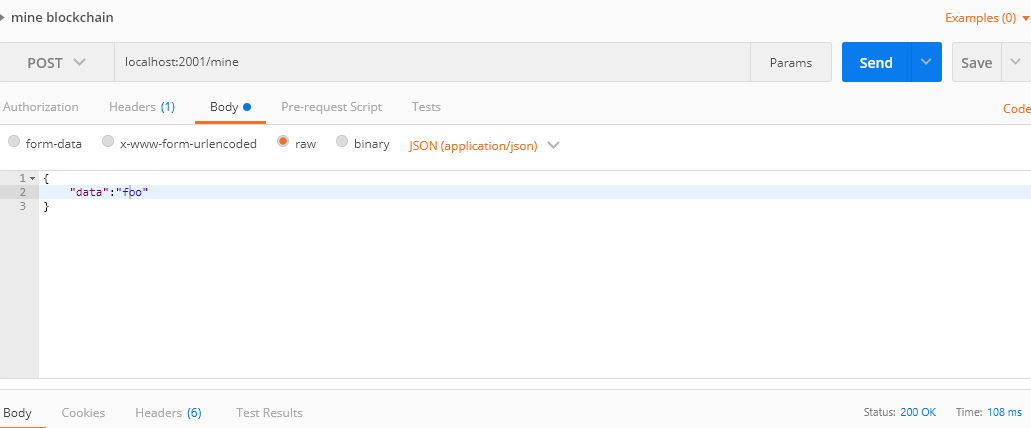
Add the messaging between the web sockets

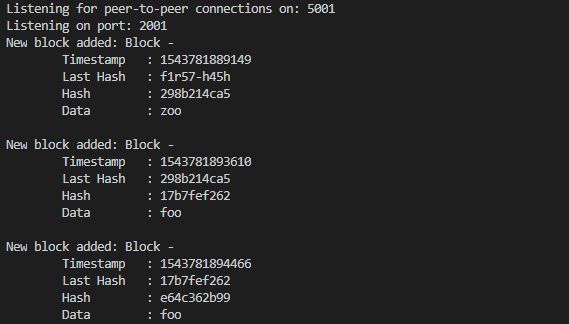
* (Kill all the running instances on the command line).
* Fire up one instance with

$ npm run dev



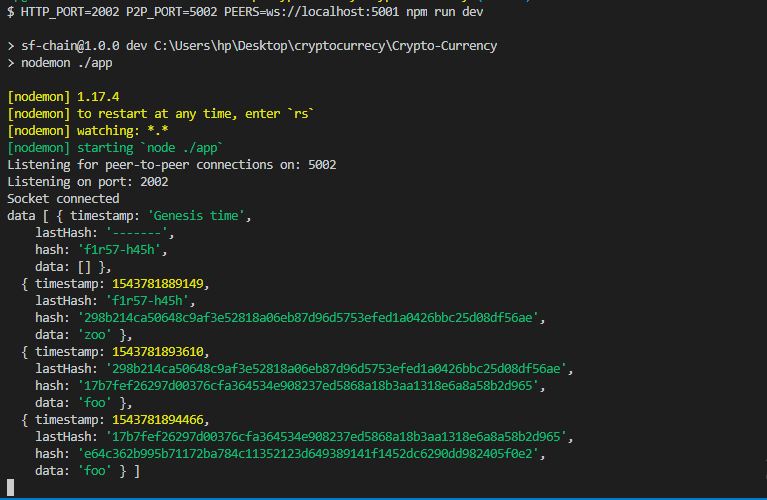
Run the postman app add some new blocks





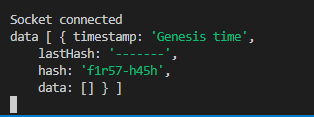
After that we would lunch a second instance now it should contain the entire block chain from other,

* Run a second instance in a second command line tab:
* $ HTTP\_PORT=2002 P2P\_PORT=5002 PEERS=ws://localhost:5001 npm run dev
* As you see it contains all the blocks from the first one.



* Observe the received message - the blockchain of the original instance.

Now check the first instance we have the messaging of the second block chain.



Synchronizing

When ever new block is added to the chain we want out

Use the received chain to synchronize chains across all instances with the `replaceChain` function. In the `P2pServer` class, in the `messageHandler` function:

* messageHandler(socket){
* socket.on('message', message => {
* // convert Stringfy to JSON
* const data = JSON.parse(message);
* console.log('data', data);
* // represent pain from an other chain
* this.blockchain.replaceChain(data);
* })

}

When ever new block is added to a chain we want each of our pairs to be aware of that new block. We want the peers to be updated each time. For this case we will implement a function called We added synchronizing function to the block chain and named it syncChain.

The goal of this function will be sending the block chain of the current instance to all of the socket pairs. For this we will use a forEach function that will run in each of our sockets.

A function sendChain that will take a socket as a parameter and send the block as Json.

sendChain(socket){

socket.send(JSON.stringify(this.blockchain.chain));

}

connectSocket(socket){

this.sockets.push(socket);

console.log('Socket connected');

this.messageHandler(socket);

*// sendChain*

this.sendChain(socket)

}

*// synchroize the change*

syncChain(){

this.sockets.forEach(socket =>this.sendChain(socket));

}

In the index.js call the p2pserver syncChain

*//Post Request*

app.post('/mine', (req, res) => {

const block =bc.addBlock(req.body.data);

console.log(`New block added: ${block.toString()}`);

*// Update the chains*

p2pServer.syncChain();

*//redirect to the*

res.redirect('/blocks')

});

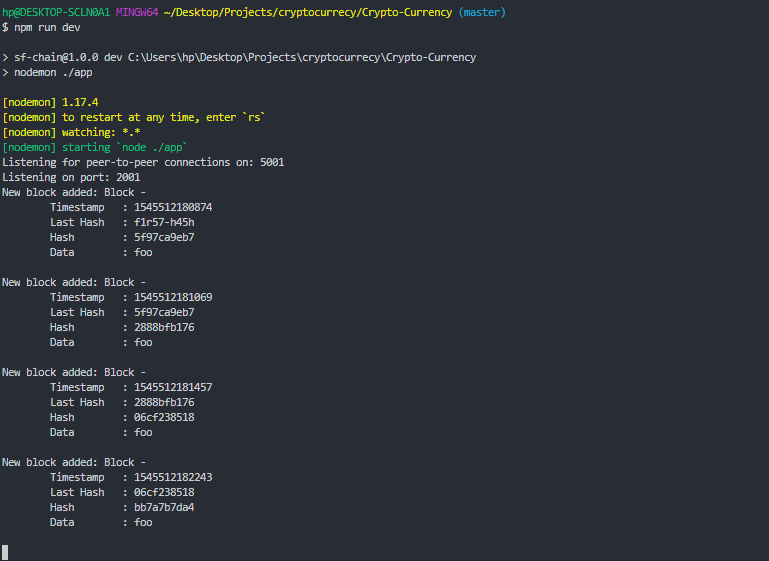
Now when one individual update the others also receive the update.

To Test:

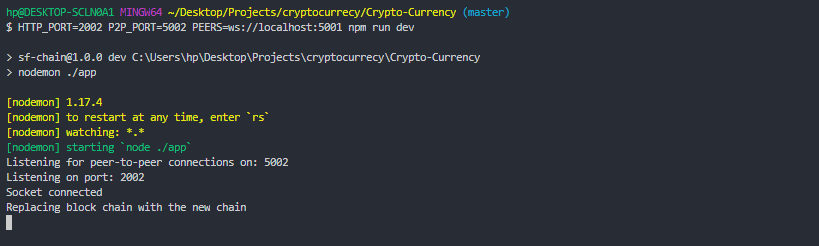
Lunch command prompt and run

Npm run dev

And mine some blocks

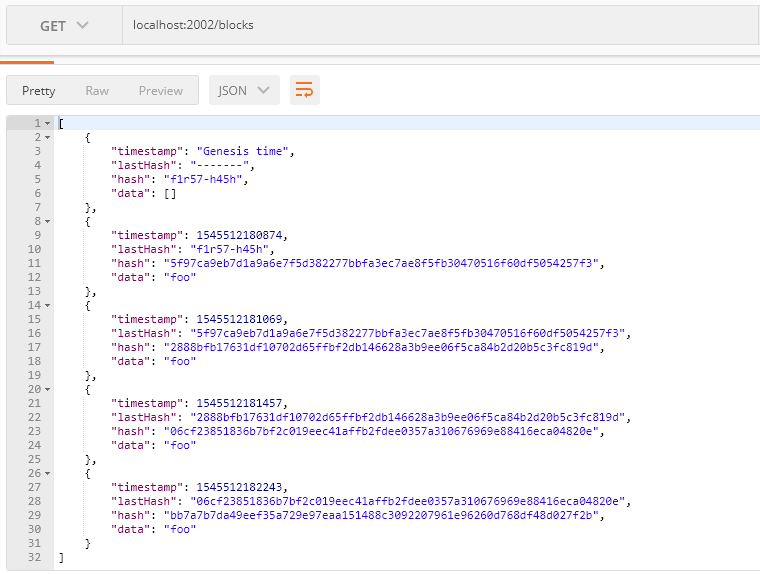


HTTP\_PORT=2002 P2P\_PORT=5002 PEERS=ws://localhost:5001 npm run dev



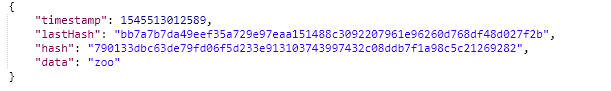
A socket connected for peer to peer receiving. We are seeing the replacing the blockchain with the new chain. It got the chain from the first instance since that chain was longer and was valid it replace your own chain. It updated itself to be in sync to the longest chain in the system.

We can confirm this by check the blocks of the second end point via postman.

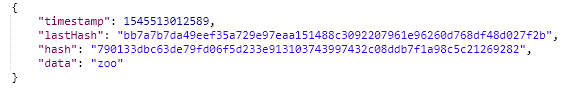


Notice that this is the chain we created our first instance. With this we have chain replacement.

Now lets mine a new block on the first end point



Lets check the second end point



We can see the second end point is also synchronized.

Proof of Work system

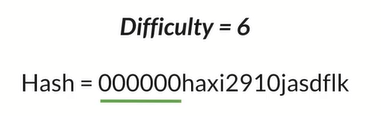
It is a system that requires a miner to spend time on doing computational work, in order to add blocks to the chain. In our decentralized block chain any peer has the capability of submitting a new chain to the system, as long as that chain is long enough, it contains valid hash data, starting from the gensis block. That chain will be accepted by all peers in the block chain network.

The proof of work system makes it expensive to generate corrupt chains. This system makes individial nodes the block chain can and submit manageable amount of computational work in order to add a block.

Many crypto currencies block chain such as bitcoin uses the proof of work system inspired by a system called hashcash. It was created in 1997. It was used to prevent email spamming as well as denial service attacks that over load web servers.

A proof of work system inspired by Hashcash work like this.

At any given point there is a diffuclty level in the block chain system. Depending on this difficulty we would mice try to add a new block we will have add a hash value for this block that matches this difficulty for this difficulty mince have to find the same number of leading zeros as the current difficulty for the generated hash of the new block to add to the chain finding the leading zeros randomly becomes exponentially harder as the difficulty rises.



In order to solve the proof of work a miner will have to generate tons of hashes in order to find one that satisfies the level of difficulty (Generate hashes until a one with the matching leading 0’s is found).

They will generate the new hashes for the new block based on the block data and the adjusting value “nonce”. sNonce is a value for generating the new hashes.

By changing the nonce value the miner generates new valid hashes for the current block and its data.

Nonce is a term in cryptograph to refer to a value that can be used only once a value for generating the new hashes. Each unique nonce will generate each unique hash.

The nonce value starts from zero and increment upward, untill a nonce is used that has a matching number of the leading zeros according to set difficulty while generating the hash.

After, nonce value is stored as a part of the block. The act of generating new hash with changing nonce value takes a computational work. The computational works spend to add a block to the chain is called mining.

Once a miner has successfully mined a block they will submit their blocks with the found nonce values to other miners. After the miner knows this nonce value they can verify the validity of that solution and add the new block to the chain.

That way they don’t have to redo the same computational work. Once they receive a new block to add to their own block chain and stay update.

By setting difficulty, block chain also have the ability to control the rate of adding new block to the systems. It means the higher the difficulty, the longer it will take for the miners to add a new block, likewise the lower the difficulty the less time it will take for miners to add(mine) a block.

For example: bitcoin sets the rate to a new block around every 10 minutes. As more miners are added to the network a more computational power is devoted to the system. The higher the difficulty of solving the proof of work algorithm becomes. The constant adjustment of this difficulty setting keeps a rate of blocks added to the bitcoin at a steady rate of every 10 minutes.

This lead us to an interesting scenario called 51% Attack

51%Attack

Is a scenario where a dishonest miner has more than at least 51% of the network’s power. Thus, they would have the power to replace the current block chain with one in their favorite. With this computing power they can generate a long enough block chain that sloved enough proof of work puzzles in order to generate another block chain that everyone would have accept. How ever the proof of work system make the situation very computational expensive that it is absolutely ridiculous to spend that cost for manuplating the block chain. Where a honest miner.

For example to take the bitcoin network at this time in all of it connected computing power is 6 billion dolars

*A 51% attack for bitcoin would be more than $6 billion (start 2018)*

How would the attacker gain the control over this system? Over all the proof work system turns dishonest miner in the block chain network and it also help us to control the rates which new blocks are added to the chain.

**Implementing Proof of work algorithm**

Adding the nonce value

The value that is used for hash calculation, by chaning the nonce values miner can generate the new hash values with a number of leading zeros that match the current difficulty of the block chain system.

We will declare the difficulty in block.js

*//require from the crypto-js*

*//Generate a unique hash based on the provide data*

const SHA256=require('crypto-js/sha256')

*//level of difficulty*

const DIFFICULTY=4;

class Block{

*//we call attributes that a block need*

constructor(timestamp, lastHash, hash, data, nonce){

*//input given to this constructor bind to the class*

this.timestamp = timestamp;

this.lastHash = lastHash;

this.hash = hash;

this.data = data;

this.nonce = nonce

}

*//will be using for debugging (in OOP)*

*//for last hash 10 digit is enough*

toString(){

return `Block -

Timestamp : ${this.timestamp}

Last Hash : ${this.lastHash.substring(0,10)}

Hash : ${this.hash.substring(0, 10)}

Nonce : ${this.nonce}

Data : ${this.data}

`

}

static genesis(){

*//this- return itself (zero default nonce value)*

*//last hash dmmy value, a dummy value, empty array*

return new this('Genesis time','-------','f1r57-h45h', [], 0)

}

static mineBlock(lastBlock, data){

*//generate a timestamp*

let hash, timestamp;

const

*//local hash constant assign to the hash of the last block*

const lastHash= lastBlock.hash;

let nonce = 0;

*// generate hash each iteration*

*// from zero up to difficulty and while condition run as long as this hash not equal zero. repeat we run*

do{

nonce++;

*// Generate timestamp at each iteration*

timestamp = Date.now();

hash = Block.hash(timestamp, lastHash, data, nonce);

}while(hash.substring(0, DIFFICULTY) !=='0'.repeat(DIFFICULTY));

*// check of the proof of work algorithm*

*//return the new instance of the block (it is same as genesis)*

return new this (timestamp,lastHash, hash, data, nonce);

}

*//static hash function: represent the unique data that we want to generate the hash for*

*//Providing argument the time stamp of the block, last hash and the data the block is storing*

static hash(timestamp,lastHash, data, nonce){

*//combing as one string using ES6 string*

return SHA256(`${timestamp} ${lastHash}${data}${nonce}`).toString();

*//replace the to do with hash*

}

*//for generating hashes*

static blockHash(block){

*//all values to generate hash*

const {timestamp, lastHash, data, nonce}= block;

return Block.hash(timestamp, lastHash, data, nonce);

}

}

*//file is shared among others*

module.exports=Block;

We added the nonce value and Difficulty Level

**Test the Nonce Functionality**

To store constants we use a config.js and store the Difficulty

And add the third block test

*//3.Test*

it('generates a hash that matches the difficulty', ()=>{

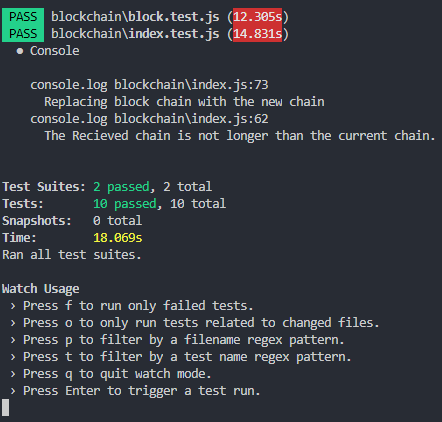
expect(block.hash.substring(0, DIFFICULTY).toEqual('0').repeat(DIFFICULTY));

});

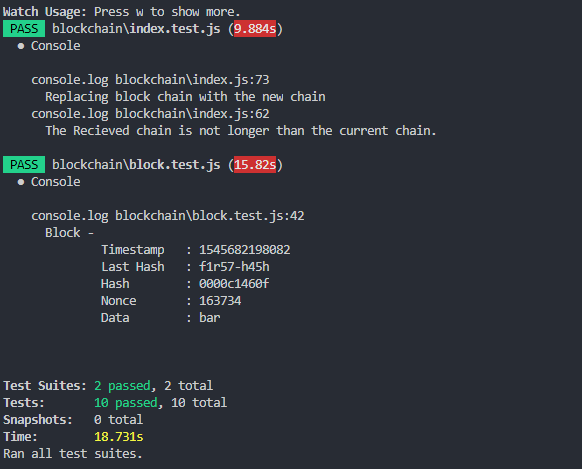
And open the command prompt

Npm run test

It will take a short moment to generate a block



We verified the nonce value incremented. We can speed up the process by speeding up the Difficulty.



Now we can see the nonce value. And we print the block.

If difficulty is the same always it will always take the same time for generating the new block. When the difficulty change dynamically it will change the generating new block time too.

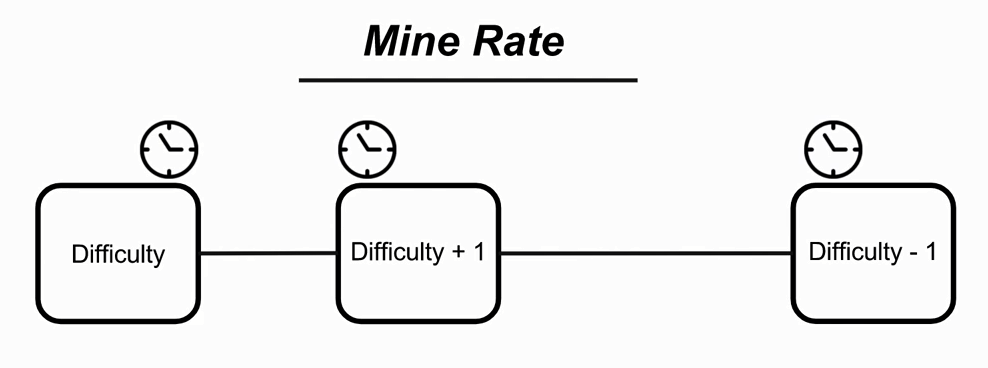
**Dynamically Block Difficulty**

We implemented a proof of work system, where miners must spend computational power to add blocks to the chain. However, as more pairs are adding blocks to the chain block will be discovered at a faster rate. As there will be a higher chance for at least one miner to discover a valid hash at the set difficulty, therefore we need a dynamic system that adjusts the blockchain difficulty level as more miner added to the system.

To achieve this we can add a difficulty attribute to each block, and set a time value called mine rate which will represent the time for each block to be mined.

Difficulty Adjust Mechanism

We will check the timestamp of newly mined block and compare to the timestamp of previously minned block. If the difference between the both timestamp is lower than a mine rate then we know that the block was mined very difficultly and we will raise the difficulty by 1. Likewise if the difference in timestamps between newest block and the previous block is bigger than a mine rate, we know that the block was mined very slowly, therefore we will lower the difficulty by 1.



With this system we will always adjust the difficulty for each new block. We should have a system that mine blocks at set interval through mine rate.

**Code the system.**

Place the MINE\_RATE in config.js

*//level of difficulty*

const DIFFICULTY = 4;

*//level of mine rate (in mili-sec)*

const MINE\_RATE = 3000;

module.exports = { DIFFICULTY, MINE\_RATE };

We need to adjust the difficulty level dynamically; The difficulty of each block will be the difficulty of the previous block.

In the block.js

const SHA256=require('crypto-js/sha256')

const { DIFFICULTY , MINE\_RATE} = require('./config')

class Block{

constructor(timestamp, lastHash, hash, data, nonce, difficulty){

*//input given to this constructor bind to the class*

this.timestamp = timestamp;

this.lastHash = lastHash;

this.hash = hash;

this.data = data;

this.nonce = nonce;

this.difficulty = difficulty || DIFFICULTY;

}

*//for last hash 10 digit is enough*

toString(){

return `Block -

Timestamp : ${this.timestamp}

Last Hash : ${this.lastHash.substring(0,10)}

Hash : ${this.hash.substring(0, 10)}

Nonce : ${this.nonce}

Difficulty : ${this.difficulty}

Data : ${this.data}

`

}

static genesis(){

*//this- return itself (zero default nonce value)*

*//last hash dmmy value, a dummy value, empty array (zero starting nonce)*

return new this('Genesis time','-------','f1r57-h45h', [], 0, DIFFICULTY);

}

static mineBlock(lastBlock, data){

*//generate a timestamp*

let hash, timestamp;

*//local hash constant assign to the hash of the last block*

const lastHash= lastBlock.hash;

*//last block difficulty for the new block difficulty*

let {difficulty} = lastBlock

let nonce = 0;

*// generate hash each iteration*

*// from zero up to difficulty and while condition run as long as this hash not equal zero. repeat we run*

do{

nonce++;

*// Generate timestamp at each iteration*

timestamp = Date.now();

difficulty = Block.adjustDifficulty(lastBlock, timestamp);

hash = Block.hash(timestamp, lastHash, data, nonce, difficulty);

}while (hash.substring(0, difficulty) !== '0'.repeat(difficulty));

*// check of the proof of work algorithm*

*//return the new instance of the block (it is same as genesis)*

return new this (timestamp,lastHash, hash, data, nonce, difficulty);

}

*//static hash function: represent the unique data that we want to generate the hash for*

*//Providing argument the time stamp of the block, last hash and the data the block is storing*

static hash(timestamp,lastHash, data, nonce, difficulty){

*//combing as one string using ES6 string*

return SHA256(`${timestamp} ${lastHash}${data}${nonce}${difficulty}`).toString();

*//replace the to do with hash*

}

*//for generating hashes*

static blockHash(block){

*//all values to generate hash*

const {timestamp, lastHash, data, nonce, difficulty}= block;

return Block.hash(timestamp, lastHash, data, nonce, difficulty);

}

*//adjust difficulty*

static adjustDifficulty(lastBlock, currentTime){

let { difficulty } = lastBlock;

difficulty = lastBlock.timestamp + MINE\_RATE > currentTime ? difficulty + 1 : difficulty - 1;

return difficulty;

}

}

*//file is shared among others*

module.exports=Block;

In adjust Difficulty we adjust the difficulty as explain before.

**Test Difficulty Adjustment**

*//4.Test*

it('lowers the difficulty for slowly mined blocks', ()=>{

*// input*

expect(Block.adjustDifficulty(block, block.timestamp+36000)).toEqual(block.difficulty-1);

});

And test open the command prompt

Npm run test

We will get an error this time we need to update the test 3. DIFFICULTY with last block difficulty level

*//3.Test*

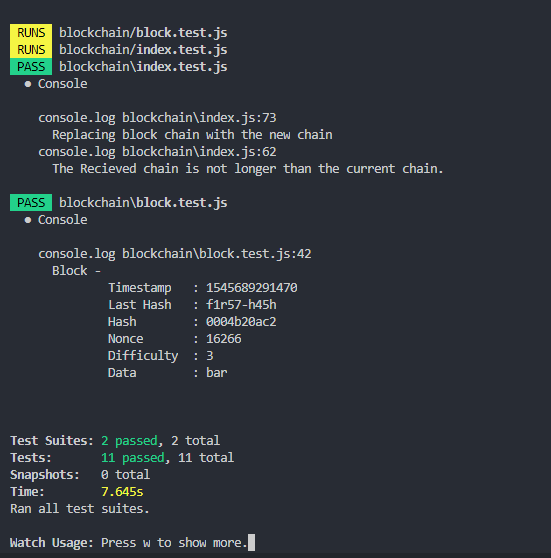
it('generates a hash that matches the difficulty', ()=>{

expect(block.hash.substring(0, block.difficulty)).toEqual('0'.repeat(block.difficulty));

console.log(block.toString());

});

Output



Let s add the other test for quickly mined blocks

*//5.Test*

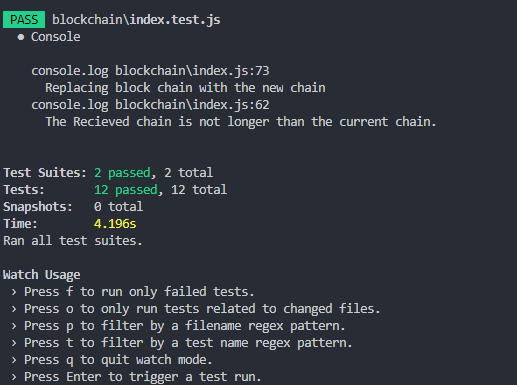
it('raises the difficulty for quickly mined blocks', ()=>{

*// input*

expect(Block.adjustDifficulty(block, block.timestamp+1)).toEqual(block.difficulty+1);

});

Output:



dev-test.js we will write a test that add 10 block in block chain.

It will take around 30 second to generate according to the mine rate which is 3 seconds.

const Blockchain = require('./blockchain');

const bc = new Blockchain();

*// add 10 blocks*

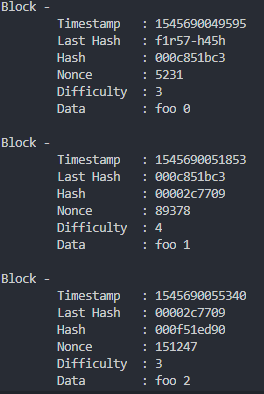
for (let i = 0; i<10 ; i++){

console.log(bc.addBlock(`foo ${i}`).toString());

}

Now run this script

Npm run dev-test



See it automatically adjust the difficulty level. When the difficulty is high it will take time suppose a bitcoint uses 6 minutes you can see takes time to generate a single coin.

We completed the proof of work system with this demonstration in place and completed the tests. We can determine the attackers replacing the block chain with the corrupt chain in their favor since it will take a lot of computational power.