Funcionalidad 00

Crear entrada en Blockchain

Para esta función he precisado de dos clases inicialmente, para instanciar la blockchain. La clase Bloque, que emula un bloque de la blockchain y la propia blockchain. Sobre todo está hecho por comodidad a la hora de dar la forma a la blockchain y tener métodos para comprobar la validez de la blockchain.

Las clases son las siguientes:

Clase Bloque

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

class Block {

constructor(index, timestamp, logroPin, data, hashPrev = "") {

this.index = index;

this.hash = this.calcularHash();

this.timestamp = timestamp;

this.logroPin = logroPin;

this.data = data;

this.hashPrev = hashPrev;

}

calcularHash() {

return SHA256(this.index + this.timestamp + JSON.stringify(this.logroPin) + JSON.stringify(this.data) + this.hashPrev).toString();

}

}

module.exports = Block

En esta clase vamos a referenciar todos y cada uno de los bloques de la blockchain. Lo primero que se puede observar es que tenemos un import. Este nos sirve de cara al algoritmo de cifrado de nuestra información. Se usa el SHA256, que consiste en a través de una cadena string con los datos a cifrar de cualquier tamaño, transformar en otra cadena de tamaño fijo, 256bits, de ahí su nombre. Esta cadena recibe el nombre de hash. Este algoritmo es muy susceptible a cualquier mínimo cambio en los datos. Si un dato es cambiado, aunque sea un carácter, el hash no será el mismo.

El constructor del bloque contiene las variables necesarias: index, timestamp, logroPin, data y hasPrev. La variable index se usa para referenciar el número del bloque. Timestamp es una marca de tiempo, para saber cuando se creó el bloque. LogroPin es la información con nuestro NFT. Data es el campo donde irán nuestras transacciones, las que se hagan en todo momento en el bloque. En nuestro caso, solo cabe lugar a dos. Creación y traspaso del LogroPin. Y por último tenemos el hashPrev, que es para hacer el anidamiento entre bloques. Este campo será rellenado con el hash del bloque anterior. En caso de ser el primer bloque será rellenado con un valor por default: “0”.

Y quedaría en el interior del constructor la variable hash. En esta variable haremos uso del algoritmo SHA256. Llamaremos a la función calcularHash(), y en ella, devolveremos la variable que alberga el import, con la especialidad de que, acompañándola de paréntesis y pasándole todos los datos menos el hash entre dichos paréntesis. El toString() del final, nos permitirá dejar el return en forma de String

Ahora es el turno de la clase Blockchain:

const Block = require('./block')

class Blockchain {

constructor() {

this.chain = [];

}

setChain(chainImport) {

this.chain = chainImport

}

getChain() {

return this.chain

}

lastIndex() {

return this.chain.length

}

getLastBlock() {

return this.chain[this.chain.length - 1]

}

addBlock(newBlock) {

if (this.lastIndex < 1) {

newBlock.hashPrev = "0"

} else {

newBlock.hashPrev = this.getLastBlock().hash

}

newBlock.hash = newBlock.calcularHash()

this.chain.push(newBlock)

}

isBlockchainValid() {

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

const currentBlock = this.chain[i];

const previousBlock = this.chain[i - 1];

if (currentBlock.hash != currentBlock.calcularHash()) {

return false;

}

if (currentBlock.hashPrev != previousBlock.hash) {

return false;

}

}

return true;

}

}module.exports = Blockchain

El import, nos sirve para poder tener acceso a los métodos de la anterior clase creada. El constructor de la clase alberga un array que almacenará la blockchain. Inicialmente está vacío. Los métodos getChain() y setChain(), sirven de cara a cargar la información de la blockchain y para cambiar la información de la blockchain.

La función lastIndex() nos sirve para obtener el tamaño actual de la blockchain, basta con llamar a chain.length. La función getLastBlock(), devuelve el último bloque de la cadena existente.

Las dos últimas funciones son: addBlock(newBlock) y isBlockchainValid(). La función addBlock(), nos sirve para añadir un bloque nuevo a la blockchain. Con el para bastaría con la sentencia this.chain.push(newBlock). Pero tenemos que rellenar hashPrev y hash. Si es el primer bloque de la cadena, no tendrá hashPrev, entonces de ahí la comprobación previa antes de asignar el hashPrev al bloque Y despues obviamente calcular de nuevo el hash del bloque actual y así, obtener el bloque. Y isBlockchainValid() comprueba la validez de la Blockchain. Hay dos parámetros importantísimos para comprobar la validez de la cadena: el hash estar bien calculado y que el hashPrev del bloque actual coincida, con el hash del bloque anterior. Lo que hacemos en este método es, comprobar a partir del segundo bloque que se cumple en este momento en el blockchain. Guardando bloque actual y bloque anterior.

Una vez elaboradas las clases, será momento de ir armando la blockchain en el servidor. Al usar node.js.

const express = require('express')

const app = express()

const port = 3000

var routes = require('./routes/blockchain');

var mantenimineto = require('./routes/mantenimiento');

app.use(express.json())

app.use('/', routes)

app.use('/mantenimiento', mantenimineto)

app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

})

Este es el archivos server.js, cada vez que queramos ejecutar su contenido, tendremos que poner en un terminal node server.js en la carpeta del proyecto. El contenido del mismo consta de:

* **express** → esta variable nos sirve para albergar el import del módulo express. Este módulo sirve para hacer servidores de manera sencilla.
* **app** → esta variable nos sirve para llamar a la anterior y referenciarla cada vez que queramos llamar al servidor
* **port** → variable que albergará el puerto de escucha para el servidor
* **routers y mantenimiento** → variable que alberga el archivo js, con la correspondiente información en dicho archivo.
* **app.use** → Cuando hablamos de rutas dentro del servidor, nos referimos a las diferentes urls/direcciones que pueda tener nuestro servidor. La aplicación con el primer parámetro podrá manejar solicitudes a lo que pongamos en esta parte izquierda de la función. A la derecha diferenciaremos las variables de los archivos.js anteriormente mencionados. Por ejemplo, si tenemos app.use('/', routes) y app.use('/mantenimiento', mantenimineto). A los métodos que tengamos en routes, con poner localhost:3000/funcióndentrodeblockchain.js nos permitiran ejecutarlos o por ejemplo en el caso de mantenimiento, sería añadir localhost:3000/mantenimiento/funcióndentrodeblockchain.js. Así es como funciona el direccionamiento.
* **app.listen** → Función por la cual, podemos hacer que el servidor escuche las peticiones, pasandole el puerto y elaborando una función para ver la correcta ejecución. Por decirlo de alguna manera, es donde se arranca el servidor.

Así acaba la elaboración del server.js, ahora iremos a los respectivos archivos js:

const Block = require('../block')

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

var models = require('../models');

var express = require('express');

const controllerDB = require('../contracts/controllerDatabase');

var router = express.Router();

router.get('/', (req, res) => {

res.send('Buenas!')

});

router.get('/crearNuevoNFT', async function(req, res) {

let simCoin = new Blockchain();

simCoin.chain = await controllerDB.allBlocks(res);

simCoin.addBlock(new Block(simCoin.lastIndex(), new Date(), { nombre: "El pistolero 7", descripcion: "El llanero solitario 7", image: "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg", owner: "1" }, { amount: 1 }))

console.log(JSON.stringify(simCoin, null, 4))

console.log(simCoin.getLastBlock())

controllerDB.createBlock(simCoin, res)

});

module.exports = router;

const Block = require('../block') → Variable para albergar el contenido de la clase Block

const Blockchain = require('../blockchain') → Variable para albergar el contenido de la clase Blockchain

var models = require('../models'); → Variable para albergar el contenido de la carpeta models. Todos y cada uno de los models para sequelize.

var express = require('express'); → Variable para el servidor

const controllerDB = require('../contracts/controllerDatabase'); → Variable para los métodos a la DB

var router = express.Router(); → Variable para inicializar todo lo referente a las rutas/direccionamiento.

Todo esto se exporta a router, la variable del server.js. Para las otras rutas creadas en server.js, se debería poner su correspondiente variable

router.get('/', (req, res) => {

res.send('Buenas!')

});

router.get('/crearNuevoNFT', async function(req, res) {

let simCoin = new Blockchain();

simCoin.chain = await controllerDB.allBlocks(res);

simCoin.addBlock(new Block(simCoin.lastIndex(), new Date(), { nombre: "El pistolero 7", descripcion: "El llanero solitario 7", image: "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg", owner: "1" }, { amount: 1 }))

//console.log(JSON.stringify(simCoin, null, 4))

//console.log(simCoin.getLastBlock())

controllerDB.createBlock(simCoin, res)

});

router.get(‘/’) y router.get('/crearNuevoNFT'), son las rutas de las que hablábamos antes, poner “localhost:3000/función/”. Aquí es donde se usan y se definen estas “funciones”. La forma de llevarlo a cabo es por métodos HTTP clásicos. Si no hay necesidad de cambiar datos ni nada, lo suyo es usar get. La ruta con “/” sin más, es el directorio por defecto. poner “localhost:3000”. Si lo ponemos, aparecerá un Buenas! en pantalla. Para la otra ruta, tendremos que poner “localhost:3000/creaNuevoNFT”, de está manera accederemos a esta función. La función es async ya que las peticiones a la base, de manera normal, no devuelven nada a esta clase. simCoin es la blockchain instanciada, el siguiente paso es rellenar la cadena con el método allBlock de controllerDB. Ahora la blockchain está llena con los diferentes bloques de la tabla Blockchain. Añadimos el nuevo bloque a la blockchain con la correspondiente información y por último llamamos al método createBlock del controllerDB, pasándole la cadena.

Ahora vamos con todo lo referente a controller y models. En cuanto a los models, crearemos:

module.exports = (sequelize, DataTypes) => {

const Blockchain = sequelize.define('Blockchain', {

index: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: DataTypes.INTEGER

},

timestamp: {

allowNull: false,

type: DataTypes.DATE

},

logroPin: {

allowNull: null,

type: DataTypes.JSON

},

data: {

allowNull: true,

type: DataTypes.STRING

},

hash: {

allowNull: false,

type: DataTypes.STRING

},

hashPrev: {

allowNull: false,

type: DataTypes.STRING

},

}, {

createdAt: false,

updatedAt: false

});

Blockchain.associate = function(models) {

};

return Blockchain;

};

Esta clase nos permite crear la Tabla blockchain en nuestra base. Elegimos los tipos que queramos para cada columna, las referencias a otras tablas si las hay y por último retornamos el objeto.

Para el controller, usaremos una cosa como esta:

const Sequelize = require('sequelize');

const db = require('../models')

module.exports = {

async allBlocks(res) {

return db.Blockchain.findAll();

},

createBlock(req, res) {

return db.Blockchain.create({

index: req.lastIndex() - 1,

timestamp: Date.now(),

//Mirar si es mejor guardar en DB como array o como String - Ambas funcionan bien

logroPin: req.getLastBlock().logroPin,

data: JSON.stringify(req.getLastBlock().data),

hash: req.getLastBlock().hash,

hashPrev: req.getLastBlock().hashPrev

})

.then(blockchain => res.status(200).send(blockchain))

.catch(error => res.status(400).send(error))

}

}

const Sequelize = require('sequelize'); → Variable por sí para alguna query se necesita de sequeñile

const db = require('../models') → variable para los modelos de la carpeta models

Con module exports puesto de esta manera podemos cada vez que en otras clases llamaremos a controllerDB, llamar a estos métodos. allBlocks(), lo que hace es llamar al modelo de Blockchian y pide que se se busque todos los elementos de dicha tabla (findAll).

createBlock nos permite la creación de una entrada en la tabla Blockchain de nuestra DB. Rellenamos toda la información que se necesite. Y será creado en la base.