Node.js server

Básico, es crear un simple una aplicación simple e ir evolucionandola en función de lo que queramos. Hecha en Node, podremos crear nuestro servidor y con los diferentes modulos añadiendo opciones de escucha, de hacer operaciones, de ruteo,...

BBDD

Lo primero que hay que hacer es un esquema de lo que queremos tener. Las tablas básicas Usuarios, Logropins y Monederos. Lo siguiente serían los registros que queremos guardar en cada una de las tablas. Mi idea es la siguiente:

| Tabla | Registros | | | |
| --- | --- | --- | --- | --- |
| Usuarios | id | Nombre | Apellido |  |
| Logropins | id | AddressToken | TokenName | UserId\* |
| Monederos | id | LogroPines | Puntos | UserId\* |

\*Estas claves, son foreing keys pertenecientes a otras tablas. Nombre intuitivo.

Pueden ser estos valores o los que se deseen. Por el momento decido que sean estos.

Por esa parte estaría hecho, ahora bien, para la parte de implementación, hemos dicho que se emplea sequelize y mysql para el manejo de eventos para la DDBB. ¿Pero cómo se hace que funcione todo esto? De la siguiente manera:

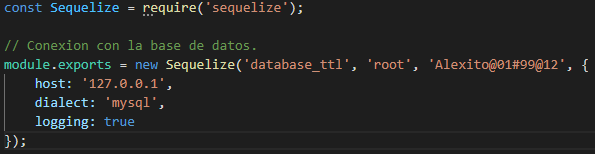
Lo primero es que una vez creado nuestro proyecto node, usemos mediante terminal los siguientes comandos siguientes: “npm install -g sequelize-cli” y “npm install -g sequelize-mysql2”. Así instalaremos los node\_modules necesarios para programar con Sequelize.

Una vez realizado este paso, nos dispondremos a introducir nuevamente otro comando: “sequelize init”, este comando nos permite crear directorios dentro de nuestro proyecto para trabajar con sequelize. Una vez introducido veremos, que hay nuevas carpetas, en concreto config, que será la cuál alberga el archivo de configuración para conectar con nuestra base de datos.

Si le echamos un vistazo, tendremos la siguiente información:



Podemos ver que podemos configurar libremente usuario, contraseña, nombre de la base, la dirección host de la misma y el dialecto que usa la base de datos. Tanto para testeo, desarrollo y producción. También hay otra forma, la cual he usado para permitirme hacer comprobaciones antes de usar la base, la cual es la siguiente:



Aquí podemos observar que es parecido al caso anterior, lo único en este caso estamos creando un objeto con la conexión, mismos parámetros que en el anterior caso, pero añadiendo logging, para las SQL query logging.

Dicho esto, podremos empezar a tocar los models, que es por decirlo así, la forma de crear nuestros objetos de cada tabla de la base. Todos los archivos en esta carpeta, llevarán la extensión .js. El archivo comienza siempre de la misma forma:

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

};

En el interior de la función, es donde crearemos el objeto. Por ejemplo, crearemos el objeto para la tabla Usuarios. La forma para declarar el objeto Usuarios es la siguiente:

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

}, {});

Y dentro de esta variable introduciremos los diferentes valores/keys y propiedades que tenga la tabla Usuario en nuestro caso. Por ejemplo, sin relacionar tablas entre sí, este objeto podría quedar así:

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

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: DataTypes.INTEGER

},

username: {

allowNull: false,

type: DataTypes.STRING

},

fullSurname: {

allowNull: false,

type: DataTypes.STRING

}

}, {});

* allowNull: Admitir valores nulos o no
* autoIncrement: Autoincremento del valor
* primaryKey: Ser primary key o no
* type: Tipo de valor

Y existen más tipos de propiedades, solo hay que saber lo que se necesita. El siguiente paso serían las asociaciones. Aquellos valores que puedan venir de otra tabla. Aquí tenemos la opción de crear foreing keys de nuestra base.

En nuestro proyecto, las tablas que tenemos son las siguientes. Usuarios, monederos y logropines. Las relaciones entre las tablas tendrían que ser:

* Un usuario tiene un monedero y un monedero tiene un usuario → 1:1
* Un usuario puede adquirir varios logropines, pero solo un logropin es adquirido por un usuario → 1:N
* Un monedero puede tener varios logropines, pero un logropin solo puede estar enlazado a un monedero → 1:N

Para establecer esta relaciones, tenemos funciones que nos permiten llevarlas a cabo sin necesidad de crear nosotros las tablas correspondientes y son las siguientes:

* hasOne(): Para las relaciones 1:1
* belongsTo(): Para las relaciones 1:1
* hasMany(): Para las relaciones 1:N
* belongsToMany(): Para las relaciones N:M

La asociación A.hasOne(B) significa que existe una relación uno a uno entre A y B, con la clave externa definida en el modelo de destino (B). La asociación A.belongsTo(B) significa que existe una relación uno a uno entre A y B, con la clave externa definida en el modelo de origen (A). La asociación A.hasMany (B) significa que existe una relación de uno a varios entre A y B, con la clave externa definida en el modelo de destino (B). La asociación A.belongsToMany (B, {mediante: 'C'}) significa que existe una relación de muchos a muchos entre A y B, utilizando la tabla C como tabla de unión.

La forma de llevarlo a cabo, es la siguiente:

Objeto.associate = function(models) {

Objeto1.hasMany(models.Objeto2, {//opciones key})

};

Siguiendo todo lo comentado anteriormente, podemos crear los keys de las tablas. Quedaría de la siguiente manera:

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

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

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: DataTypes.INTEGER,

},

address: {

allowNull: false,

type: DataTypes.STRING

},

money: {

allowNull: false,

type: DataTypes.STRING

},

idsLogroPines: {

allowNull: true,

type: DataTypes.STRING

}

}, {});

Monederos.associate = function(models) {

Monederos.hasMany(models.Logropines)

};

return Monederos;

};

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

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

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: DataTypes.INTEGER,

},

nameLP: {

type: DataTypes.STRING,

},

addressLP: {

type: DataTypes.STRING,

},

}, {});

Logropines.associate = function(models) {

};

return Logropines;

};

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

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

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: DataTypes.INTEGER

},

username: {

allowNull: false,

type: DataTypes.STRING

},

fullSurname: {

allowNull: false,

type: DataTypes.STRING

}

}, {});

Usuarios.associate = function(models) {

Usuarios.hasOne(models.Monederos);

Usuarios.hasOne(models.Logropines);

};

return Usuarios;

};

Con esto, ya tendríamos la parte de los objetos terminada. Ahora sería el turno de configurar el servidor, para aceptar peticiones para poder conectar con la base y llevar a cabo las diferentes opciones que deseemos con las tablas. ¿Cómo lo hacemos? De la siguiente manera:

Tendremos que tocar el archivo index.js de modesl, el app.js y por último crear tantos archivos como rutas queramos implementar y con rutas, nos referimos a los accesos dentro del servidor gracias al node\_module “Router”.

Comenzamos con el archivo en models: index.js. Este archivo es quien montará la conexión y guardará todos los objetos mientras estemos trabajando en el servidor. El archivo es autogenerado cuando introducimos el comando “sequelize init”, asi que no hay que tocar mucho de este. Yo le hice un pequeño cambio para poder comprobar la conexión con la base, al inicio del servidor. El archivo me quedo de la siguiente manera:



Mi cambio es la variable sequelize, que venía de una manera distinta a la que se presenta en la imagen. A parte de que es algo más cómodo, no me permitía hacer la comprobación en app.js y de esta manera, sí que podía lograrlo. Este archivo se guarda en una variable llamada db, la cual se podrá usar donde se le llame.

El siguiente archivo es app.js. Este es el archivo principal del servidor. Es quien iniciará las comunicaciones y quién proveerá de las diferentes rutas para el mismo. Las rutas por el momento las he dividido en:

* usuario → Crear, eliminar, modificar
* monedero → Crear, eliminar, modificar
* logropine → Crear, eliminar, modificar
* mantenimiento → Configuración base, crear base, eliminar base y crear de nuevo

Cuando llamemos a localhost:puerto/usuario por ejemplo, podremos acceder a los diferentes métodos que elijamos en dicha ruta. Y esto con todos y cada una de las rutas mencionadas anteriormente. ¿Cómo lo hacemos en código? Así quedaría app.js:

var express = require('express');

var app = express();

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

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

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

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

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

app.use(express.json());

app.use('/', routes);

app.use('/usuario', usuarios);

app.use('/monedero', monederos);

app.use('/logropin', logropines);

app.use('/mantenimiento', mantenimiento);

app.listen(3000, function() {

console.log('Iniciado el servicio backend de Training to Learn, en el puerto 3000');

});

De esta manera, podremos proveer al servidor de las rutas. Mencione anteriormente, que quería hacer una comprobación de la conexión con la DB. Pues bastará con rellenar con las siguientes líneas, en el código anterior:

var express = require('express');

var app = express();

const db = require('./config/database.js'); <---

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

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

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

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

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

//Comprobar que la conexion es correcta

db <---

.authenticate() <---

.then(() => { <---

console.log('Connection has been established successfully.');<---

}) <---

.catch(err => { <---

console.error('Unable to connect to the database:', err); <---

}); <---

// Rutas

app.use(express.json());

app.use('/', routes);

app.use('/usuario', usuarios);

app.use('/monedero', monederos);

app.use('/logropin', logropines);

app.use('/mantenimiento', mantenimiento);

app.listen(3000, function() {

console.log('Iniciado el servicio backend de Training to Learn, en el puerto 3000');

});

Y así, ya hemos finalizado app.js, solo tendremos que tocarlo alguna vez más si decidimos añadir más rutas.

Ahora para finalizar este apartado, lo que tendremos que hacer es ir a la carpeta de routes y crear los correspondientes archivos .js para las rutas seleccionadas. Algo así tendría que ser:



Y el contenido de cada archivo variará en función de lo que queramos hacer. Vamos a elegir Usuarios para explicar lo que tendríamos que hacer en cada uno de los archivos.

En Usuarios, por ejemplo, queremos hacer las acciones básicas: Crear, modificar y eliminar.

Lo primero es lo que venimos haciendo en otros archivos, aquí lo importante es models, ya que estamos llamando a los objetos que tenemos en /models.

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

var express = require('express');

var router = express.Router();

Lo siguiente es ir llamando a “Router.” acompañado del método HTTP que necesitemos para la query. En mi caso, he usado POST para todos. Lo que cambia es el contenido de cada uno, sería una cosa así:

router.post('/crear', function(req, res) {

models.Usuarios.create({

id: req.body.id,

username: req.body.username,

fullSurname: req.body.fullSurname,

}).then(() => {

res.json({ ok: true });

}).catch((val) => {

res.json({ ok: false, error: val });

});

});

router.post('/eliminar', function(req, res) {

models.Usuarios.destroy({

where: {

id: req.body.id

}

}).then(() => {

res.json({ ok: true });

}).catch((val) => {

res.json({ ok: false, error: val.name });

});

});

router.post('/modificar', function(req, res) {

models.Usuarios.update({

id: req.body.idNuevo,

username: req.body.usernameNuevo,

fullSurname: req.body.fullSurnameNuevo,

}, {

where: {

id: req.body.id

}

}).then(() => {

res.json({ ok: true });

}).catch((val) => {

res.json({ ok: false, error: val.name });

});

});

module.exports = router;

Gracias a Sequelize, tenemos métodos propios para crear, modificar y eliminar un objeto. Bastaría con llamar a models.Usuarios.create/update/destroy. Así podremos hacerlo. Dentro de esta función, haremos la query, y es un poco especial, ya que no es como las habitales. En caso de crear o modificar, obtendremos los valores que vienen por el body de la petición HTTP (como aparece en la imagen) y a continuación, vemos si ha habido problemas y si los ha habido, lo notificamos (métodos .then y .catch). El where como bien sabido es un condicionante para las querys, ya que buscará lo que pidamos en este apartado. Este lo usaremos tanto para update, después de recoger los datos del body y por último en el delete, ya que buscaremos al usuario a modificar/eliminar por su id.

Y esta, sería la forma de operar en todos los siguientes archivos.

Aunque he probado a reducir el número de dependencias para este módulo. Quitando todas las rutas para usuario, monedero y logropines. Es el archivo index, quién ejecutará todas las opciones y al que le llegarán los datos también. Mantenimiento se queda, ya que es importante de cara a rellenar datos, resetear la base de datos,...

Blockchain

En cuanto a la documentación sobre cómo se puede crear tu propia blockchain, hay múltiples formas de llevarlo a cabo. La mayor parte de documentación, no habla para hacerlo de manera permanente, si no, que se busca hacerlo todo en local y de manera secuencial por iteración. La idea de este proyecto es que no sea así. Tiene que ser información permanente y que se pueda acceder a ella cuando queramos.

Mi idea, es, creando las clases básicas ser capaz de crear un servidor que albergue la blockchain. Puede ser conjunto con el servicio backend o puede ser por separado en otro puerto.

Lo primero, es ver como funciona aunque sea de manera secuencial, la blockchain. Lo primero, es saber que es una blockchain: Es una cadena de bloques, entonces ya tenemos nuestras primeras 2 clases: Blockchain y Block, una para referirse a la cadena y otro para referirse al bloque.

La clase Block

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

class Block{

constructor(index, timestamp, logroPin ,data, ownerAddress, 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 + this.hashPrev + this.hash+JSON.stringify(this.logroPin)+JSON.stringify(this.data)).toString();

}

}

module.exports = Block

Lo primero que observamos es la variable SHA256. Es nuestro algoritmo de encriptado de la información para la blockchain. Es uno de los puntos fuertes que tiene, la seguridad. Y se la brindamos al sistema con este cifrado. Solo tenemos que llamar a la liberia crypto-js/sha256 y lo tendremos.

Al ser la clase Block, tendremos que crear un constructor, con toda la información que deseemos introducir en cada bloque. En este caso, tendremos que ver qué información nos interesa para cada bloque. Mi idea es la siguiente:

* index → El número del bloque
* hash → Es un string de longitud fija, es creada a partir de los datos e independientemente de la longitud de los mismos, seguirá teniendo el mismo tamaño.
* timestamp → Una marca de tiempo, una fecha
* logroPin → Los datos de nuestro LogroPin en la blockchain
* data → Aquí es un campo en el que podrían ir todos los datos que se quisieran, por ejemplo las diferentes transacciones, de que A paso a B x dinero.
* hashPrev → Cada bloque debe conocer su anterior, y el identificador por excelencia es el hash, entonces albergaremos el hash del bloque anterior al actual.

Por el momento veo estas como posibles candidatos a los elementos para el constructor.

La función calcularHash(), es la función encargada de calcular el algoritmo SHA256 con los datos del constructor. Cabe destacar que cada cifrado es único, ya que al mínimo cambio, dará una salida diferente. Y por último exportar a un módulo toda esta información con “module.exports”.

const Block = require('./block')

class Blockchain{

constructor(){

this.chain = [this.createInitialBlock()];

}

createInitialBlock(){

var date = new Date()

var dateFormat = date.getDate() + "/" + (date.getMonth()+1) + "/" + date.getFullYear();;

return new Block(0, dateFormat, {logroPin:{ nombre: "El pistolero", descripcion: "El llanero solitario", image: "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg", owner: "0x00000000001" }}, {amount : 1},"0x00000000001", "0")

}

getLastBlock(){

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

}

addBlock(newBlock){

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

La clase Blockchain, incluirá un import de la clase anteriormente explicada. Esta clase es la principal y a la que llamaremos para las diferentes operaciones de la blockchain.

Dejaremos el constructor para el final. La función createInitialBlock(), sirve para crear el bloque inicial de nuestra blockchain, el denominado: “Génesis”. Bastará con rellenar un bloque con la información justa o no, y hacer que se sigan los demás bloques a partir de este.

getLastBlock(), es una función que nos permite obtener el último bloque de la cadena. La forma es llamar a la cadena, al último elemento existente.

addBlock(newBlock), nos permite añadir un nuevo bloque a nuestra cadena. Como preparación antes de hacer push a la cadena de bloques, tendremos que decir cuál es el hash del bloque anterior y cuál será el hash bloque actual. Usamos la función anterior para poder obtener el hash del último bloque actual de la blockchain. Y para calcular el hash del nuevo bloque, llamaremos a calcularHash()

isBlockchainValid(), aquí validamos si la cadena de bloques es correcta y no hay fallos entre eliminaciones de algún bloque o modificación de la información de algún bloque. Lo que se hace es muy simple, se va bloque por bloque comprobando la información referente a los hash. Se empieza desde el segundo bloque, ya que el primer bloque no tiene un prevHash. Aunque se podría retocar la función de tal manera que si estamos en el primer bloque, podamos comprobarlo de manera diferente. Ahora vamos con la explicación:

Se guardan el bloque actual y el anterior, y se comprueba por un lado, que el hash del actual sea el mismo que llamara a la función de calcular Hash. Y por otro lado, comprobar que están bien unidos los bloques mediantes hash y hashPrev de sus bloques respectivos.

Y así tendríamos lo básico para la blockchain, ahora solo quedaría probar toda esta información.

let simCoin = new Blockchain();

var date = new Date()

var dateFormat = date.getDate() + "/" + (date.getMonth()+1) + "/" + date.getFullYear();;

simCoin.addBlock(new Block(1, dateFormat, {logroPin:{nombre: "El pistolero 2", descripcion: "El llanero solitario 2", image: "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg", owner: "0x00000000002", }}, {amount : 1}, "0x00000000002"))

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

Poniendo todo esto en un archivo app.js, añadiendo los correspondientes imports, ejecutando en un terminal node app.js, podremos ver nuestra blockchain. Nos saldría lo siguiente:

{

"chain": [

{

"index": 0,

"hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",

"timestamp": "7/11/2021",

"logroPin": {

"logroPin": {

"nombre": "El pistolero",

"descripcion": "El llanero solitario",

"image": "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg",

"owner": "0x00000000001"

}

},

"data": {

"amount": 1

},

"ownerAddress": "0x00000000001",

"hashPrev": "0"

},

{

"index": 1,

"hash": "6e833781ff6c6763c6bff169cc9a482cccab934fad57136c8a1b745b0b5a8572",

"timestamp": "7/11/2021",

"logroPin": {

"logroPin": {

"nombre": "El pistolero 2",

"descripcion": "El llanero solitario 2",

"image": "https://pbs.twimg.com/profile\_images/1244332617652211715/R6MEhciZ.jpg",

"owner": "0x00000000002"

}

},

"data": {

"amount": 1

},

"ownerAddress": "0x00000000002",

"hashPrev": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

}

]

}

Hasta aquí todo bien, ahora se viene el lugar donde hay poca información/documentación. Esto solo saldrá por lo que pongamos en el app.js. No hay persistencia de los datos, ¿Cómo hacemos que sea persistente?

No hay mucha documentación acerca de este aspecto y la que hay gira en torno a Ethereum, no una blockchain propia.

La idea principal es montar el servidor node.js para poder por un lado tener información persistente y que pueda añadirse información nueva y por otro lado realizar las diferentes funciones con los contratos inteligentes.

El primer inconveniente es la búsqueda de una forma de almacenar toda la información de la blockchain. Al ser toda la información en forma de JSON lo suyo sería guardarlo en un json. Aunque no se cuanto de seguro es dicha forma de almacenar la información. Si fuera más complicada esta opción, o por temas de seguridad no se pudiera, se podría usar el método clásico, base de datos. Mi duda sería enlazar los datos entre las tablas.

El segundo problema sería reestructurar las clases, ya que varias cosas no tienen sentido para una estructura con persistencia. En concreto sería la estructura de blockchain. Habría que buscar una manera, para que al llamar al constructor, cargarse toda la información de golpe. Si es con una base de datos, es muy sencillo, ya que solo es conectar la base con la aplicación, con el json sería algo más laborioso, como leer el documento, recoger la información, escribir en el momento que se desee, guardar y repetir.

Teniendo esto, acompañado de node.js, tendríamos nuestra estructura completada, al menos en este apartado, ya que la última parte sería, la realización de contratos inteligentes, que es otra parte complicada. Ya que los contratos ya creados como OpenZeppelin son exclusivos de Ethereum, y siguen las normas de Ethereum. No he visto ningun sitio en el que ponga que se puedan utilizar en redes privadas que no sean de Ethereum, que sean de algo propio.

Para este apartado he pensado en la realización de una clase Contrato, la cual se encargue de realizar las funciones que normalmente hace un contrato de OpenZeppellin. Crear coleccionable, verificar autenticidad, traspasar,...

Lo suyo sería que con los diferentes métodos hiciera comprobaciones pertinentes con la blockchain. Por ejemplo, a la hora de hacer un traspaso, comprobar que el material que se va a traspasar es el dueño quien lo va a traspasar,...

Es una cosa que aún estoy investigando. Pero la idea es esa. Y una vez tengamos esta parte, lo suyo sería hacer uso del contrato en el módulo de creación del LogroPin

---------------------------------------------------------------------------------------------------------------------------

Volvemos a la parte del Blockchain. Cabe destacar que en los múltiples tutoriales que hay por el internet tenemos un sistema de recompensas por minado de bloques, sistema de transacciones y luego si se lleva más lejos una validación de firma para las transacciones. Como estamos hablando de que queremos que el coste sea el menor posible, hablando incluso de coste 0, así que la idea para el blockchain es la siguiente.

* Va a haber usuario normales y luego los canjeadores (las personas que permiten el canje de LogroPins por puntos). El intercambio solo se puede dar entre ambas entidades, nunca entre dos del mismo tipo.
* El canjeador tendrá que tener puntos de sobra para distribuir cuando un usuario quiera canjear su LogroPin por puntos para su monedero
* Para mi no tiene sentido un sistema de recompensa de minado, si solo va a ser un red de intercambio. No creo que nadie se ponga a minar bloques para obtener recompensas. Aquí hay dos algoritmos:
  + **Proof of Work** es un algoritmo de consenso común utilizado por las redes de criptomonedas más populares como Bitcoin y Litecoin. Básicamente, el Proof of Work requiere que se instale una copia de la cadena de bloques completa en una máquina local. Con cada nueva transacción, el blockchain (la cadena de bloques) compara los datos de cada máquina y la transacción será aprobada si los datos coinciden en el 51% de los casos. Esto requiere muchos cálculos, potencia de cálculo y electricidad (en otras palabras, requiere mucho trabajo), por lo que se llama Proof of Work.
  + **Proof of Stake** es otro algoritmo de consenso común, pero en lugar de utilizar la potencia de las máquinas de minería, utiliza las monedas de los usuarios apostadas en carteras congeladas para verificar las transacciones. En Proof of Work, cuanta más potencia de cálculo poseas, más ventaja tendrás en la minería, mientras que en Proof of Stake, la probabilidad de minar un criptoactivo depende de la cuota de monedas (stake) que tengas. Por ejemplo, alguien que posee el 5% de todas las monedas creará alrededor del 5% de los nuevos bloques.

Si hubiera que utilizar alguno, sería el proof of work, pero no creo que se dieran recompensas, ni se dejarán bloques pendientes. Se hace el algoritmo y punto.

* Las transacciones tienen que estar si o si, son las que nos dirán en cada bloque, que operaciones se han hecho
* La firma, depende de por donde se quiera llevar, pueden ser crypto wallets reales o con imaginarias. Con las imaginarias debería bastar con una búsqueda en la base de datos, ver si es el dueño y si hay que pagar, comprobar que tiene la cantidad de dinero suficiente para el canje

Una vez hablado de estos puntos, procedemos a hablar sobre cada uno de los diferentes problemas que surgen:

Almacenaje

Para poder guardar los datos, he visto que existe la posibilidad de poder almacenarlos también en una base de datos. Vamos a rehacer esta parte por completo, pero aprovechando cosas que ya hicimos en su momento.