Node.js notas:

# Parte I

1. requires(./nombre\_archivo) esto para importar librerías
2. exports.nombre = función; hay que exportar las funciones que serán llamadas por el requires
3. crear un objeto const Math = {};
4. exportar ese objeto  
     
   Math.add = add;

Math.substract = substract;

Math.multiply = multiply;

Math.divide = divide;

module.exports = Math;

1. Documentación <https://nodejs.org/dist/latest-v16.x/docs/api/>
2. Llamado de modulo pre instalados: const os = require('os');
3. Modulo sistema operativo  
   const os = require('os');

console.log(os.platform());

console.log(os.release());

console.log('free mem', os.freemem());

1. Manejo de archivos:  
     
   const fs = require('fs');

fs.writeFile('./texto.txt', 'la persona que me gusta es: ', function(err){ // ubicacion del archivo, mensaje a escribir, funcion callback para aber el error

    if(err){

       console.log(err);

    }

    else{

        console.log('No se habla de bruno nonono');

    }

});

console.log('ultima linea de codigo');

fs.readFile('./texto.txt',  function(err,data){ // ubicacion del archivo,  funcion callback para saber el error o leer lo datos

    if(err){

       console.log(err);

    }

    else{

        console.log(data.toString());

    } });

1. http

const http = require('http');

http.createServer(function (req, res ){ // peticion request req, respuesta response res

res.writeHead(200, {'content-type': 'text/html'});// cabecera, e pone un código de estado http y el contenido de lo //////que se se espera (texto html)

res.write('<h1>Hola mundo desde node js</h1>'); // escibir

res.end();

}).listen(3000);

10)otro modo:  
  
const handleserver = function (req, res ){ // peticion request req, respuesta response res

res.writeHead(200, {'content-type': 'text/html'});// cabecera, e pone un codigo de estado http y el contenido de lo que se se espera (texto html)

res.write('<h1>Hola mundo desde node js</h1>'); // escibir

res.end();

}

const server = http.createServer(handleserver);

server.listen(3000, function(){// crea una funncion callback despues de acceder al servidor

console.log('server on port 3000');

});

11)usando npm  
  
EN la consola escribir

npm install colors

12) uso colors   
const color = require('colors');

console.log('server on port 3000', .green);

13) npm init // sirve para hacer una descripción del curso

Crea un package.jon

14) dentro del paquete json como decirle cual es el sitio inicial:

"scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "node index.js"

luego puedo iniciar el proyecto con el codigo npm start

15)correr otra categoría del script npm run nombre de la categoría creada

16)instalar express

npm i express

"dependencies": {

    "colors": "^1.4.0",

    "express": "^4.17.2"

  },

17)mismo codigo con express

const express = require('express');

const server  = express();

server.get('/', function(req, res){

     res.send('<h1>Hola MUndo</h1>');

     res.end();

});

server.listen(3000,function(){

    console.log('server on port 3000'.green)

} );

# PARTE 2: EXPRESS

1. routing

server.get('/about', (req,res)=>{ // creando la ruta about

    res.send('About me');

});

1. npm install nodemon: framework que me permite introducer cambios inmediatos a node reiniciar el servidor  
     
   uso de nodemon: npx nodemon index.js
2. permitir que el servidor use json

server.use(express.json()); // hace que express entienda el formato json

4)método post(dinámico)

Si por ejemplo a parte de /user s quiere agregar algo mas (ejemplo <http://localhost:5000/user/456>)

server.post('/user/:id',(req,res)=>{ //mostando lo recibido del post en consola

    console.log(req.body);

    console.log(req.params);

    res.send('REQUEST POST');

});

5)

Acceder a los parámetros de la URL

server.put('/aput/:id',(req,res)=>{

    console.log(req.body);

    res.send('User ${req.params.id} updated'); // para acceder al parametro en la url

});

6)

server.all('/user', (req,res,next)=>{

    console.log('por aqui paso');

    next();

 });

Ojo este “all” tendría que estar de primero”

# Parte 3 Middleware

1) )los middleware funcionan para procesar datos antes de llegar a la ruta

function logger(req,res,next){ //middleware

    console.log('Request Recieved');

    next();

}

Uso:  
  
server.use(logger);

2) para saber el protocolo (http, https,ftb) el host (ip, dirección, localhost) y el puerto:

console.log( `Request Recieved: ${req.protocol}://${req.get('host')}${req.originalUrl}`);

el middleware debe ser definido antes de las rutas

3) Morgan

Sirve para dar la información del request

const morgan = require('morgan'); //llamado modulo morgan npm i morgan

uso

server.use(morgan('dev'));

respuesta:

GET /user 304 6.534 ms - -

4) static files (se usa por defecto en caso que no se acceda a ninguna ruta)

server.use(express.static('public'));// uso de express static en caso de que no se acceda a ninguna ruta

la carpeta public debe tiene un index.html que accede a una carpeta css o cualquier otro archivo javascript que se acceda por defecto

5) java script timer

setTimeout(() => {

    document.getElementById('loquesea').innerHTML = 'Cambiado desde inner html';

}, 3000);

# Parte IV settings

1)

// todo esto se usa en la parte donde el puerto escucha

//palabras reservadas get y set

server.set('AppName','TutoNode'); // nombre de la app

server.set('port', 3000); // puerto de uso

uso:

server.listen(server.get('port'),function(){

    console.log(server.get( 'AppName')); // nombre de la app puesto en settings, debe estar olo

    console.log('server on port  '.green, server.get('port'));

} );

2) plantillas

Ejs en un framework para poder ejecutar plantillas de vitas en web

Injstalacion npm install ejs

Debe existir una carpeta de views con un archivo “.ejs” el cual tiene la vista que se quiere visualizar

Uso desde las rutas:

//routes

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

const data = [{name: 'olga'},{name: 'ruben'},{name: 'joey'},{name: 'joey'}]

    res.render('index.ejs');

});

Imprimir la plantilla desde el archivo .ejs

<body>

<h1>EJS</h1>

<ul>

    <% people.forEach(function (person){ %>

        <li>

            <%= person.name  %>

        </li>

    <% }) %>

</ul>

Hay otras plantillas como pug pero honestamente creo que los frameworks frontend hacen esto también

# Parte 4 database

1. <http://expressjs.com/en/guide/database-integration.html#postgresql> en este sitio hay varias formas de conectarse a una base de datos, hay que ter en cuenta que las bases de datos tienen sintaxis diferentes por lo que se recomienda ORM para unificarlos en código y objetos javascript
2. ORM: convierte los query en objetos javascript para ser leídos ejemplos sequelize y mongoose
3. Instalar sequalize

npm install --save sequelize

1. Instalar drivers, aqui hay varios ejemplos, usaremos el de postgresql porque ya tengo una tabla en esa bd

# One of the following:

$ npm install --save pg pg-hstore # Postgres

$ npm install --save mysql2

$ npm install --save mariadb

$ npm install --save sqlite3

$ npm install --save tedious # Microsoft SQL Server

1. Conectar a la base de datos, hay 3 formas…. Yo eligire la forma 1 de momento

const { Sequelize } = require('sequelize');

// Option 1: Passing a connection URI

const sequelize = new Sequelize('sqlite::memory:') // Example for sqlite

const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname') // Example for postgres

// Option 2: Passing parameters separately (sqlite)

const sequelize = new Sequelize({

dialect: 'sqlite',

storage: 'path/to/database.sqlite'

});

// Option 3: Passing parameters separately (other dialects)

const sequelize = new Sequelize('database', 'username', 'password', {

host: 'localhost',

dialect: /\* one of 'mysql' | 'mariadb' | 'postgres' | 'mssql' \*/

});

1. Probar la conexión

try {

await sequelize.authenticate();

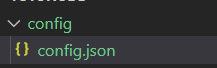
console.log('Connection has been established successfully.');

} catch (error) {

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

}

1. Forma mas cool! Utilizar sequalize cli  
     
   en la terminal escribir:  
     
   npm install -g sequelize-cli
2. Para usar la CLI  
     
   npx sequelize init  
     
   creara muchas cosas:
3. Primero una carpeta config



Donde se ve info como esta:  
  
Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Aquí es donde se debe buscar la info de la base de datos, deben asegurarse que donde dice dialect se ponga la base de datos que se va a usar, en este caso postgresª

1. Los modelos:



en este archivo se ve todo lo relacionado con tomar los datos de la base de datos

11) crear bd

Usar el comando npx sequelize db:créate

12) crear tabla

npx sequelize model:generate –name nombre\_tabla User –attributes campo1:tipo, campo2:tipo, campon:tipo

13) llamado en el index.js (servidor)

const {sequelize} = require('./models')

async function main(){

    await sequelize.sync();

}

main();

13) cambiae el nombre de la tabla:

En el javascript generado para la tabla ir a la sección de User.init() y agregar:

User.init({

    name: DataTypes.STRING,

    email: DataTypes.STRING,

    role: DataTypes.STRING

  }, {

    sequelize,

    tableName: 'users', // cambio de tabla

    modelName: 'User',

  });

  return User;

14) agregar mas propiedades a los datos: nuevamente en user.init hacer la siguiente modificación(comparar con (13) )

 User.init({

    name: {

      type: DataTypes.STRING,

      allowNull: false,

    },

    email: {

      type: DataTypes.STRING,

      allowNull: false,

    },

    role: {

      type: DataTypes.STRING,

      allowNull: false,

    },

  }, {

    sequelize,

    tableNme: 'users',

    modelName: 'User',

  });

  return User;

necesario volver al index.js principal del servidor y forzar a hacer los cambios en la base de datos:

async function main(){

    await sequelize.sync({force: true}); // necesario para que surjan los cambios en bd

}

main();

15) acceder a la base de datos desde un post

server.post('/user',async(req,res)=>{ //mostando lo recibido del post en consola

    console.log(req.body);

    //console.log(req.params);

    const {name, email,route} = req.body

    res.send('REQUEST POST');

    try{

        const user = await User.create({name, name,role});

        return res.json(user);

    }

    catch(err){

           console.log(err);

           return res.status(500).json(err);

    }

});

Quitar la función main creada y mover el await para el server.listen

server.listen(server.get('port'),async function(){

    await sequelize.sync({force: true}); // necesario para que surjan los cambios en bd

    console.log(server.get( 'AppName')); // nombre de la app puesto en settings, debe estar olo

    console.log('server on port  '.green, server.get('port'));

    console.log('Database synced! ' )

} );

16) UUID en la base de datos

uuid: {

      type: DataTypes.UUID,

      defaultValue: DataTypes.UUIDV4,

      allowNull: false,

    },

17) migraciones: para no borrar toda la base de datos usamos migraciones

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Existen dos funciones en migrations up and down, cambiar al nombre de la tablename (punto 13)

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Estas funciones up and down crean y eliminan las tablas respectivamente

En el archivo de migración hay que poner el tipo de dato uuide creado decir que no se permite allow null

module.exports = {

  async up(queryInterface, DataTypes) {

    await queryInterface.createTable('usuarios', {

      id: {

        allowNull: false,

        autoIncrement: true,

        primaryKey: true,

        type: DataTypes.INTEGER

      },

      uuid: {

        type: DataTypes.UUID,

        defaultValue: DataTypes.UUIDV4

      },

      name: {

        type: DataTypes.STRING,

        allowNull: false

      },

      email: {

        type: DataTypes.STRING,

        allowNull: false

      },

      role: {

        type: DataTypes.STRING,

        allowNull: false

      },

      createdAt: {

        allowNull: false,

        type: DataTypes.DATE

      },

      updatedAt: {

        allowNull: false,

        type: DataTypes.DATE

      }

    });

  },

  async down(queryInterface, DataTypes) {

    await queryInterface.dropTable('usuarios');

  }

};

18) borrar la base de datos y volverla a crear

19)ahora en el servidor podemos cambiar la función synced por authenticated y ya no tenemos que forzar el reinicio de la base de datos

server.listen(server.get('port'),async function(){

    await sequelize.authenticate(); // YA NO TENEMOS QUE FORZAR EL CAMBIO EN LA BD

    console.log(server.get( 'AppName')); // nombre de la app puesto en settings, debe estar olo

    console.log('server on port  '.green, server.get('port'));

    console.log('Database synced! ' )

} );

20) para crear los datos ahora en la terminal escribimos npx sequelize db:migrate para llamar la migración

21) Esta sección es exclusiva para evitar campos respetidos

22) esta para borrar filas

23) esta para editar datos

24)ocultar datos al hacer post (quedan en la base de datos)

///////////////ocultar un dato al hacer post (queda en la db)/////////////

    toJSON(){

        return {...this.get(),id: undefined}

    }

    ///////////////////////////

Texto

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

25) mostrar un listado de todos los usuarios de la base de dtoas usando el servidor

server.get('/user', async (req,res)=>{ // retornando un json

    try{

        const usuarios = await User.findAll();

        return res.json(usuarios);

    }

    catch(err){

        console.log(err);

        return res.status(500).json({error: 'se pifio'});

    }

});

Resultado (postman)

Al hacer un request a <http://localhost:3000/user> :

[

    {

        "uuid": "11108991-6214-44f6-910c-56575dc0d4cb",

        "name": "Carl",

        "email": "elnegro@gmail.com",

        "role": "el negro supongo",

        "createdAt": "2022-02-01T13:39:50.766Z",

        "updatedAt": "2022-02-01T13:39:50.766Z"

    },

    {

        "uuid": "c51b9ff4-3598-4029-a015-1dca7cd3880f",

        "name": "lenny",

        "email": "elnoviodecarl@gmail.com",

        "role": "novio de carl",

        "createdAt": "2022-02-01T13:53:21.820Z",

        "updatedAt": "2022-02-01T13:53:21.820Z"

    }

]

26) retornar un dato único (findone)

server.get('/user/:uuid', async (req,res)=>{ // retornando un json

    const uuid = req.params.uuid

    try{

        const usuarios = await User.findOne({

            where: {uuid }

        });

        return res.json(usuarios);

    }

    catch(err){

        console.log(err);

        return res.status(500).json({error: 'se pifio en el de uno solo'.red});

    }

});

27) crear un nuevo modelo: en consola escribir  
  
npx sequelize model:generate --name nombre\_modelo --attributes columna:tipo\_de\_dato

Puede ser mas de un atributo  
  
debe realizarse lo mismo que se realizo con puntos anteriores con este modelo (datatypes, migración, etc)

28) npx sequelize db:migrate:status permite saber que esta Up y que esta down y debe crearse con sequelize db:migrate

Hacer migración de la tabla

29) Relaciones entre modelos (tablas)  
  
Cada modelo contiene una función associate la cual permite hacer las asociaciones que normalmente se harían desde el sql:

static associate({User}) { // crear relaciones entre los modelos (post -> user)

30) Relación de pertenencia 1 a 1: Si deseo dejar en claro por ejemplo que un post especifico le pertenece a un usuario

 static associate({User}) { // crear relaciones entre los modelos (post -> user)

      // definir asociacion aqui

      //userId es la llave foranea

      this.belongsTo(user,{foreignKey: 'userId', }) // este {post} pertenece a un {usuario}

    }

31) relación de pertenencia de 1 a n: si por el contrario deseo decir que un usuario puede tener muchos post:

static associate({Post}) {

      // definir asociacion aqui

      this.hasMany( Post, {foreignKey: 'userId'}); //este usuario tiene muchos posts cuidado con las mayusculas

    }

32) Hay que crear la llave forenea userId en el modelo **post**

userId: {

        type: DataTypes.INTEGER,

        allowNull: false

      },

33) una vez se hayan creado las asociaciones debe revertirse la ultima migración para crear una nueva

Se hace con el comando npx sequelize db:migrate:undo

34) y finalmente se vuelve a hacer la migración con los campos establecidos

Debería quedar así y no debería poder ingresar un post si no tiene relación con un suario especifico porque el modelo no permite campos null

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

35) una vez migrada la base datos, ir al servidor principal, primero es agregar el modelo Post en

const {sequelize, User, Post} = require('./models')

36) finalmente se crea una ruta Post donde se hace lo siguiente:

(a) Buscar un usuario equivalente al uuid el cual se desea hacer el post

(b) a aquel usuario encontrado asociar con la llave foránea userid el post creado

Una vez se realice aquello se guarda en la tabla “posts” el post asociado a un usuario

server.post('/post', async(req,res)=>{

    const{userUuid, body} = req.body;

    try{

      // si no me molesta es porque el orm diferencia entre el modelo User y la tabla usuarios por si solita

      const user = await User.findOne({where: {uuid: userUuid}}); // (a) buscar en el request un usuario igual al userUuid

      const post = await Post.create({body,userId: user.id}); //(b) crear ese post con el usuario encontrado usuario

      return res.json(post);  //retornar el post

    }

    catch(err){

        console.log(err)

        return res.status(500).json(err);

    }

});

En el código la función Post.Create() hace el insert en la base de datos, en User.findOne se hace en query el usuario especifico

La petición y la respuesta deberían verse como:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

37) para mantener oculto el código del usuario

static associate({User}) { // crear relaciones entre los modelos (post -> user)

      // definir asociacion aqui

      //userId es la llave foranea

      this.belongsTo(User,{foreignKey: 'userId' }) // este {post} pertenece a un {usuario}

    }

    ///Vamos a ocultar el usuario (justo debajo del associate en Post.js

    toJSON(){

      return {...this.get(), id: undefined, userId: undefined }

    }

En caso de que se desee incluir todas las búsquedas asociadas a una tabla especifica se usa el campo “include”

server.get('/post', async(req,res)=>{

    try{

     const posts = await Post.findAll({include : [User]}); //buscar todos los post de un usuario

      return res.json(posts);  //retornar el post

    }

    catch(err){

        console.log('Se pifio en la funcion de mostrar '.red);

        console.log(err)

        return res.status(500).json(err);

    }

});

38) Conceptos básicos de findALL  
  
tabla.findAll() es equivalente a decir Select \* FROM tabla  
  
Entonces:

Post.findAll({

where: {

authorId: 2

}

});

// SELECT \* FROM post WHERE authorId = 2;

Post.findAll({

where: {

authorId: 12,

status: 'active'

}

});

// SELECT \* FROM post WHERE authorId = 12 AND status = 'active';

39) operador “Or” : en el punto 38 se usa una función equivalente a usar el operador “And” , también se puede especificar esto. Si fuese el caso de usar el operador “ or” este se especifica del siguiente modo:

const { Op } = require("sequelize");

Post.findAll({

where: {

[Op.or]: [

{ authorId: 12 },

{ authorId: 13 }

]

}

});

// SELECT \* FROM post WHERE authorId = 12 OR authorId = 13;

40) Destroy es equivalente a hacer Delete en un campo en la base de datos:

const { Op } = require("sequelize");

Post.destroy({

where: {

authorId: {

[Op.or]: [12, 13]

}

}

});

// DELETE FROM post WHERE authorId = 12 OR authorId = 13;

41) Listado de todos los operados de sequelize

const { Op } = require("sequelize");

Post.findAll({

where: {

[Op.and]: [{ a: 5 }, { b: 6 }], // (a = 5) AND (b = 6)

[Op.or]: [{ a: 5 }, { b: 6 }], // (a = 5) OR (b = 6)

someAttribute: {

// Basics

[Op.eq]: 3, // = 3

[Op.ne]: 20, // != 20

[Op.is]: null, // IS NULL

[Op.not]: true, // IS NOT TRUE

[Op.or]: [5, 6], // (someAttribute = 5) OR (someAttribute = 6)

// Using dialect specific column identifiers (PG in the following example):

[Op.col]: 'user.organization\_id', // = "user"."organization\_id"

// Number comparisons

[Op.gt]: 6, // > 6

[Op.gte]: 6, // >= 6

[Op.lt]: 10, // < 10

[Op.lte]: 10, // <= 10

[Op.between]: [6, 10], // BETWEEN 6 AND 10

[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15

// Other operators

[Op.all]: sequelize.literal('SELECT 1'), // > ALL (SELECT 1)

[Op.in]: [1, 2], // IN [1, 2]

[Op.notIn]: [1, 2], // NOT IN [1, 2]

[Op.like]: '%hat', // LIKE '%hat'

[Op.notLike]: '%hat', // NOT LIKE '%hat'

[Op.startsWith]: 'hat', // LIKE 'hat%'

[Op.endsWith]: 'hat', // LIKE '%hat'

[Op.substring]: 'hat', // LIKE '%hat%'

[Op.iLike]: '%hat', // ILIKE '%hat' (case insensitive) (PG only)

[Op.notILike]: '%hat', // NOT ILIKE '%hat' (PG only)

[Op.regexp]: '^[h|a|t]', // REGEXP/~ '^[h|a|t]' (MySQL/PG only)

[Op.notRegexp]: '^[h|a|t]', // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)

[Op.iRegexp]: '^[h|a|t]', // ~\* '^[h|a|t]' (PG only)

[Op.notIRegexp]: '^[h|a|t]', // !~\* '^[h|a|t]' (PG only)

[Op.any]: [2, 3], // ANY ARRAY[2, 3]::INTEGER (PG only)

[Op.match]: Sequelize.fn('to\_tsquery', 'fat & rat') // match text search for strings 'fat' and 'rat' (PG only)

// In Postgres, Op.like/Op.iLike/Op.notLike can be combined to Op.any:

[Op.like]: { [Op.any]: ['cat', 'hat'] } // LIKE ANY ARRAY['cat', 'hat']

// There are more postgres-only range operators, see below

}

}

});

42) operadores que solo pueden usarse en postgres

[Op.contains]: 2, // @> '2'::integer (PG range contains element operator)

[Op.contains]: [1, 2], // @> [1, 2) (PG range contains range operator)

[Op.contained]: [1, 2], // <@ [1, 2) (PG range is contained by operator)

[Op.overlap]: [1, 2], // && [1, 2) (PG range overlap (have points in common) operator)

[Op.adjacent]: [1, 2], // -|- [1, 2) (PG range is adjacent to operator)

[Op.strictLeft]: [1, 2], // << [1, 2) (PG range strictly left of operator)

[Op.strictRight]: [1, 2], // >> [1, 2) (PG range strictly right of operator)

[Op.noExtendRight]: [1, 2], // &< [1, 2) (PG range does not extend to the right of operator)

[Op.noExtendLeft]: [1, 2], // &> [1, 2) (PG range does not extend to the left of operator)

43) mas opciones sobre queries: <https://sequelize.org/master/manual/model-querying-basics.html>

44) include trae otros campos de la base de datos

server.get('/post', async(req,res)=>{

    try{

     const posts = await Post.findAll({include : [User]}); //buscar todos los post de un usuario

      return res.json(posts);  //retornar el post

    }

    catch(err){

        console.log('Se pifio en la funcion de mostrar '.red);

        console.log(err)

        return res.status(500).json(err);

    }

});

45) Se puede modificar el json del include:

 const posts = await Post.findAll({include : [{model : User, as: 'user'}]});

pero es importante que debe especificarse ese as: ‘user” en las asociaciones en el post.js :

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

46) validaciones: si queremos evitar cosas en un formato no deseado al ingresarlo a la base de datos

name: {

      type: DataTypes.STRING,

      allowNull: false,

      validate: { // se usa para hacer validaciones

        notNull : {msg: 'User must have a name'},

        notEmpty : {msg: 'Name must not be empty'}

      }

    },

Mas de estas validaciones en: <https://sequelize.org/master/manual/validations-and-constraints.html>