Paso a paso de cómo crear una API-RET

Tabla de contenido

[Programas requeridos 2](#_Toc64570557)

[npm 2](#_Toc64570558)

[Node.js 2](#_Toc64570559)

[mongoDB 2](#_Toc64570560)

[Notas: 2](#_Toc64570561)

[Generar un proyecto base con express-generator 3](#_Toc64570562)

[Configurar la Base de Datos 3](#_Toc64570563)

[Se crea el modelo 3](#_Toc64570564)

[Se crean las rutas 4](#_Toc64570565)

[Controladores 5](#_Toc64570566)

[Paginación 6](#_Toc64570567)

[Middleware para búsquedas individuales 7](#_Toc64570568)

[Subir imágenes a la nube con Cloudinary 9](#_Toc64570569)

[Configuración de acceso a Cloudinary 9](#_Toc64570570)

[Modelo para subir los archivos a cloudinary 9](#_Toc64570571)

[Leer archivos que vengan en la petición del cliente con express 10](#_Toc64570572)

[Subir los archivos a la nube 11](#_Toc64570573)

[Generar urls amigables (SEO) 14](#_Toc64570574)

[Manejo de parámetro - helper 16](#_Toc64570575)

## Programas requeridos

### npm

* + - Para instalar o actualizar npm ejecutamos en la consola “npm install npm -g”
    - Para saber la version en la consola “npm –version”

### Node.js

* + - Verificar la versión en la consola “node --version”
    - para instalar o actualizar se descarga desde
      * https://nodejs.org/es/download/
      * node .msi y se instala

### mongoDB

* + - Descargar mongoDB
      * https://www.mongodb.com/try/download/community
    - Descarga directa de la versión 4.4.3
      * https://fastdl.mongodb.org/windows/mongodb-windows-x86\_64-4.4.3-signed.msi
    - En la instalación verificar que no se instale como servicio, para que no se ejecute siempre que iniciamos nuestro pc
    - Una vez instalado, se debe crear en el disco "C" una carpeta llamada "data" y dentro de esta una llamada "db"
    - Ejecutar el programa mongod.exe que está ubicado en...
      * C:\Program Files\MongoDB\Server\4.4\bin
      * mongod.exe siempre debe estar ejecutándose
    - Con "mongo.exe" podemos ver las db creadas y mas

## Notas:

* -g se usa para instalar de forma global
* Para detener una ejecución (como el servidor) en la consola “ctrl + C”
* para moverse entre las carpetas en la consola “cd + la ruta
* El servidor necesitara reiniciarse cada vez que se realicen cambios, mientras programamos se realizan muchos. Así que para esto se ejecuta un demonio (nodemon) que está pendiente de cada cambio.

npm install -g nodemon

## Generar un proyecto base con express-generator

* instalar express-generator de manera global
  + npm install express-generator -g
* Generar la base del proyecto
  + Nos ubicamos en el proyecto ejecutamos “express api”
  + Donde api es el nombre de la aplicación
  + C:\Desarrollos\Nodejs-REST-productos>express api
* Se quita del proyecto lo que no se utilice
* En el package “start” utilizaremos nodemon en vez de solo node
* Ejecutamos las dependencias, nos ubicamos en la base de nuestro proyecto "api"
  + ejecutamos npm install para que se instalen todas las dependencias
  + C:\Desarrollos\Nodejs-REST-productos\api>npm install
* Para ejecutar un proyecto utilizamos “node app.js”, pero como generamos el proyecto, tenemos dentro del package el script “start” que ejecuta nuestro proyecto
  + C:\Desarrollos\Nodejs-REST-places\api>npm run start
    - En caso de que salga error con el puerto porque ya está en uso, dentro de la carpeta “bin” está el archivo “www”, allí podemos cambiar el puerto, ejemplo pasar de 3000 a (3001, 3002, 3003, …)

## Configurar la Base de Datos

* Se debe tener mongoDB ejecutándose
* Como conector para la base de datos usaremos mongoose, así que se debe instalar en nuestro proyecto
  + npm install mongoose
    - PS C:\Desarrollos\Nodejs-REST-productos\api> npm install mongoose
* Se crea una carpeta llamada “config” y dentro un archivo “database.js” en el cual se tendrá la configuración de la DB

## Se crea el modelo

const **mongoose** = **require**('mongoose');

let placeSchema = new **mongoose**.**Schema**({

    titulo: {

**type**: **String**,

        required: true

    },

**autor**: **String**,

**descripcion**: **String**,

**categoria**: **String**,

**ubicacion**: **String**

});

let **Libro** = **mongoose**.**model**('Libro', placeSchema);

module.**exports** = **Libro**;

## Se crean las rutas

* Recuerda importar la ruta y usarla, en el servidor “app.js”

const librosRouter = **require**('./routes/libros');

app.**use**('/libros', librosRouter);

* libros.js

const **express** = **require**('express');

let router = **express**.**Router**();

const librosController = **require**('../controllers/LibrosController');

router.**route**('/')

*// obtener todos los registros*

    .**get**(librosController.**index**)

*// crear un registro*

    .**post**(librosController.**create**);

router.**route**('/:id')

*// obtener un registro*

    .**get**(librosController.**show**)

*// actualizar un registro*

    .**put**(librosController.**update**)

*// eliminar un registro*

    .**delete**(librosController.**destroy**);

module.exports = router;

**Que no se olvide importar y usar la ruta**

*// const Place = require('./models/Libros');*

const placesRouter = **require**('./routes/libros');

app.**use**('/libros', placesRouter);

## Controladores

* controllers/libros.js'

*/\*\**

*\* modelo*

*\*/*

const **Libro** = **require**('../models/Libro');

function **index**(req, res) {

*// obtener todos los registros*

**Libro**.**find**({}).**then**(docs => {

            res.**json**(docs);

        }).**catch**(err => {

            console.**log**(err);

            res.**json**(err);

        })

}

function **create**(req, res) {

*// crear un registro*

**Libro**.**create**({

        titulo: req.body.titulo,

        autor: req.body.autor,

        descripcion: req.body.descripcion,

        categoria: req.body.categoria,

        ubicacion: req.body.ubicacion

    }).**then**(doc => {

        res.**json**(doc)

    }).**catch**(err => {

        console.**log**(err);

        res.**json**(err);

    })

}

function **show**(req, res) {

*// obtener un registro*

**Libro**.**findById**(req.params.id)

*// Place.findOne({})*

        .**then**(doc => {

            res.**json**(doc);

        }).**catch**(err => {

            console.**log**(err);

            res.**json**(err);

        });

}

function **update**(req, res) {

*// actualizar un registro*

    let attributes = [

        'titulo', 'autor', 'descripcion',

        'categoria', 'ubicacion'];

    let placeParams = {};

    attributes.**forEach**(attr => {

        if (**Object**.prototype.**hasOwnProperty**.**call**(req.body, attr))

            placeParams[attr] = req.body[attr];

    });

*// Place.updateOne({ '\_id': req.params.id },*

*// Place.findOneAndUpdate({ '\_id': req.params.id },*

**Libro**.**findByIdAndUpdate**(req.params.id,

        placeParams, { new: true }

    ).**then**(doc => {

        res.**json**(doc);

    }).**catch**(err => {

        console.**log**(err);

        res.**json**(err);

    });

}

function **destroy**(req, res) {

*// eliminar un registro*

**Libro**.**findByIdAndRemove**(req.params.id)

        .**then**(doc => {

            res.**json**(doc); *// res.json({});*

        }).**catch**(err => {

            console.**log**(err);

            res.**json**(err);

        });

}

*// index : index*

module.exports = { **index**, **create**, **show**, **update**, **destroy** };

*//index: index*

## Paginación

* Con mongoose-paginate

npm install mongoose-paginate

Dentro del modelo Libro.js qué importaremos mongoose-paginate

const **mongoosePaginate** = **require**('mongoose-paginate');

… debajo del esquema se agrega el plugin

placeSchema.**plugin**(**mongoosePaginate**);

En el controlador LibrosController.js, se modifica la función para obtener todos los registros

*// Libro.find({})*

**Libro**.**paginate**({}, { page: req.query.page || 1, limit: 5, sort: { '\_id': 1 } })

## Middleware para búsquedas individuales

En el controlador se crea la función (remplazar place por libro)

function **find**(req, res, next) {

**Place**.**findById**(req.params.id)

        .**then**(place => {

            req.place = place;

**next**();

        }).**catch**(err => {

            console.**log**(err);

**next**(err);

        });

}

Cambios en las funciones

function **show**(req, res) {

*// buscar un sitio*

    res.**json**(req.place);

*// show pasa solo hacer render del middleware*

}

function **destroy**(req, res) {

*// Place.findByIdAndRemove(req.params.id)*

    req.place.**remove**()

        .**then**(doc => {

            res.**json**(doc.title); *// res.json({});*

        }).**catch**(err => {

            console.**log**(err);

            res.**json**(err);

        });

}

function **update**(req, res) {

*// Place.findById(req.params.id)*

*//ejecuta los "hups"*

*//   .then(doc => {*

*//     doc.title = req.body.title;*

*//     doc.description = req.body.description;*

*//     //...*

*//     doc.save();*

*//   }).catch(err => {*

*//     console.log(err);*

*//     res.json(err);*

*//   });*

    let attributes = [

        'title', 'description', 'acceptsCreditCard',

        'openHour', 'closeHour'];

    let placeParams = {};

    attributes.**forEach**(attr => {

        if (**Object**.prototype.**hasOwnProperty**.**call**(req.body, attr))

            placeParams[attr] = req.body[attr];

    });

*// Place.updateOne({ '\_id': req.params.id },*

*// Place.findOneAndUpdate({ '\_id': req.params.id },*

*// Place.findByIdAndUpdate(req.params.id,*

*//     placeParams, { new: true }*

    req.place = **Object**.**assign**(req.place, placeParams);

    req.place.**save**(

    ).**then**(doc => {

        res.**json**(doc);

    }).**catch**(err => {

        console.**log**(err);

        res.**json**(err);

    });

}

Que no se olvide exportar el middleware y agregarlo a las rutas

router.**route**('/:id')

*// buscar un sitio*

*// :comodines*

    .**get**(PlacesController.**find**, PlacesController.**show**)

*// actualizar un registro*

    .**put**(PlacesController.**find**, PlacesController.**update**)

*// eliminar un sitio*

    .**delete**(PlacesController.**find**, PlacesController.**destroy**);

## Subir imágenes a la nube con Cloudinary

Primero se guardaran los archivos en una ruta local y después se guardaran en la nube

Se debe tener una cuenta (puede ser gratuita en Cloudinary.com)

### Configuración de acceso a Cloudinary

En la carpeta config se crea “secrets.js”

module.exports = {

    cloudinary : {

        cloud\_name : "se obtiene de Cloudinary ",

        api\_key : "se obtiene de Cloudinary",

        api\_secret : "se obtiene de Cloudinary "

    }

}

Que no se olvide que estos son datos que no se deben subir al git si este no es privado

### Modelo para subir los archivos a cloudinary

Se instala la librería de Cloudinary

PS C:\Desarrollos\Nodejs-REST-places\api> npm install cloudinary

Se crea un archivo “Uploader.js” en la carpeta de modelos

const cloudinary = **require**('cloudinary');

const secrets = **require**('../config/secrets');

*// se le pasa a cloudinary la información de acceso, el json que tenemos en los secretos*

cloudinary.**config**(secrets.cloudinary);

*// imagePath es donde se va a encontrar la imagen que se subira a la nube*

*// funcion que sube las imagenes a la nube*

module.exports = function (imagePath) {

*// una promesa porque la subida puede tardar*

    return new **Promise**((**resolve**, **reject**) => {

*// subir la imagen*

        cloudinary.uploader.**upload**(imagePath, function (result) {

            console.**log**(result);

*// la promesa se resulve con la ruta de la imagen, para agregarla al elemento img*

            if (result.secure\_url) return **resolve**(result.secure\_url);

**reject**(new **Error**('Error con Cloudinary'));

        })

    })

}

### Leer archivos que vengan en la petición del cliente con express

Actualmente se utiliza el bodyParser para leer el cuerpo de las peticiones

Se necesita leer archivos para eso utilizamos **multer**

Instalamos la librería multer

PS C:\Desarrollos\Nodejs-REST-places\api> npm install multer

Configuramos donde se guardarán los archivos **config/upload.js**

const **multer** = **require**('multer');

module.exports = **multer**({ dest: 'uploads/' });

En el controlador

*// importar configuracion*

const upload = **require**('../config/upload');

…

function **multerMiddleware**(params) {

*// upload.single para subir un solo archivo*

*// return upload.single('avatar')*

*// fields permite especificar una coleccion de nombres*

*// con los que vendran los archivos que esperamos recibir*

*// maxCount indica la cantidad de archivos maximos que se pueden subir por peticion*

    return upload.**fields**([

        { name: 'avatar', maxCount: 1 },

        { name: 'cover', maxCount: 1 }

    ])

}

…

Exportar la función **multerMiddleware**

En el controlador modificar las funciones que recibirán archivos

*// crear un sitio*

    .**post**(placesController.**multerMiddleware**(), placesController.**create**);

Hasta aquí se guardan los archivos en local y a continuación se subirán a Cloudinary

### Subir los archivos a la nube

Todo el cambio es en el controlador y se utiliza el modelo **Uploader** que es el encargado de subir los archivos

Se modifica la función create, ya que esta no será la que le responda al cliente

function **create**(req, res, next) {

*// crear un sitio*

**Place**.**create**({

        title: req.body.title,

        description: req.body.description,

        acceptsCreditCard: req.body.acceptsCreditCard,

        openHour: req.body.openHour,

        closeHour: req.body.closeHour

    }).**then**(doc => {

*// res.json(doc)*

        req.place = doc;

**next**();

    }).**catch**(err => {

*// console.log(err);*

*// res.json(err);*

**next**(err);

    });

}

*// mueve la imagen a la nube*

function **saveImage**(req, res) {

    if (req.place) {

        if (req.files && req.files.avatar) {

            const path = req.files.avatar[0].path;

**uploader**(path).**then**(result => {

                console.**log**(result);

                res.**json**(req.place);

            }).**catch**(err => {

                console.**log**(err);

                res.**json**(err);

            })

        } else {

            res.**json**({

                error: 'File not found'

            });

        }

    } else {

*// en caso de que no se pueda procesar o subir la imagen*

*// 422 no se pudo procesar la entidad*

        res.**status**(422).**json**({

            error: req.error || 'Cloud not save place'

        });

    }

}

Hasta aquí se guarda en la nube

Falta guardar en la DB la ruta a la imagen

#### Subir la imagen y guardar la ruta

En el MVC una de las reglas es que la mayor parte de la lógica debe estar en el controlador y no tanto en el modelo (controladores gordos y modelos delgados)

Por esto se moverá la lógica de la subida de los lugares a donde está la lógica de los lugares, es decir,

**Modelo Place**

const uploader = **require**('./uploader');

**…**

El siguiente código sube el avatar a la nube y guarda la ruta en la DB

*// subir el avatar imagen*

placeSchema.methods.**updateAvatar** = function (path) {

    return **uploader**(path)

        .**then**(secure\_url => this.**saveAvatarUrl**(secure\_url));

}

*// guardar en la DB la ruta del avatar*

placeSchema.methods.**saveAvatarUrl** = function (secure\_url) {

    this.avatarImage = secure\_url;

    return this.**save**();

}

En el controlador en la función saveImage cambiar uploader por la nueva función

*// uploader(path)*

            req.place.**updateAvatar**(path)

Ahora se modifica el anterior código para que se suban y guarden el avatar y el cover

Modelo Place

*/ subir los archivos imagenes y ejecuta la funcion que guarda la ruta en el registro de la DB*

placeSchema.methods.**updateImage** = function (path, imageType) {

*// primero sube la imagen a la nube*

    return **uploader**(path)

*// segundo ejecuta la funcion que guarda la ruta en la DB*

        .**then**(secure\_url => this.**saveImageUrl**(secure\_url, imageType));

}

*// guardar en la DB la ruta de esos archivos*

placeSchema.methods.**saveImageUrl** = function (secure\_url, imageType) {

    this[imageType+'Image'] = secure\_url;

    return this.**save**();

}

Controlador

*// mueve la imagen a la nube*

function **saveImage**(req, res) {

    if (req.place) {

        const files = ['avatar', 'cover'];

        promises = [];

        files.**forEach**(imageType => {

            if (req.files && req.files[imageType]) {

                const path = req.files[imageType][0].path;

                promises.**push**(req.place.**updateImage**(path, imageType));

            } else {

                console.**log**('File not found')

            }

        });

**Promise**.**all**(promises)

            .**then**(result => {

                console.**log**(result);

                res.**json**(req.place);

            }).**catch**(err => {

                console.**log**(err);

                res.**json**(err);

            })

    } else {

*// en caso de que no se pueda procesar o subir la imagen*

*// 422 no se pudo procesar la entidad*

        res.**status**(422).**json**({

            error: req.error || 'Cloud not save place'

        });

    }

}

## Generar urls amigables (SEO)

A la configuración de la DB se agrega “slug”

Se crea una carpeta llamada **plugins** y dentro un archivo **slugify.js**

En el tendrá una función que recibe un string y lo transforma a un slug

*// https://gist.github.com/mathewbyrne/1280286*

module.exports = function (text) {

    return text.**toString**().**toLowerCase**()

        .**replace**(/\s+/g, '-')           *// Replace spaces with -*

        .**replace**(/[^\w\-]+/g, '')       *// Remove all non-word chars*

        .**replace**(/\-\-+/g, '-')         *// Replace multiple - with single -*

        .**replace**(/^-+/, '')             *// Trim - from start of text*

        .**replace**(/-+$/, '');            *// Trim - from end of text*

}

Los hooks de mongoose: son funciones que se pueden enlazar con el ciclo que sigue un documento cuando se guarda, elimina o actualiza, es decir, podemos ejecutar una función antes o después que se guarde, elimine o actualice un documento.

En el modelo

const slugify = **require**('../plugins/slugify');

…

Se agrego al Schema

    slug: {

        type: **String**,

        unique: true

    }

…

*// pre = antes de guardar ejecuta esta función*

*// y el hook no termina hasta que la función next se ejecute*

*// funciones para guardar un slug unico*

placeSchema.**pre**('save', function (next) {

    if (this.slug) return **next**();

    generateSlugAndContinue.**call**(this, 0, next);

});

function **generateSlugAndContinue**(count, next) {

    this.slug = **slugify**(this.title);

    if (count != 0)

    this.slug = this.slug+"-"+count;

    Place.**validateSlugCount**(this.slug).**then**(isValid => {

        if (!isValid)

        return generateSlugAndContinue.**call**(this, count+1, next);

**next**();

    })

}

placeSchema.statics.**validateSlugCount** = function (slug) {

    return Place.**count**({  slug: slug })

        .**then**(count => {

            if (count > 0)

                return false;

            return true;

        })

}

En el controlador

se modifica la función **find**

function **find**(req, res, next) {

*// Place.findById(req.params.id)*

    Place.**findOne**({ slug: req.params.id } )

        .**then**(place => {

            req.place = place;

**next**();

        }).**catch**(err => {

            console.**log**(err);

**next**(err);

        });

}

## Manejo de parámetro - helper

Se crea una función para el manejo o construcción de parámetros de crear y actualizar un registro

Dentro de la carpeta controlador se crea “helpers.js”

*/\*\**

*\* se recibe un array de los parametros validos*

*\* se recibe un objeto, el cuerpo de la solicitud*

*\* se retorna un objeto con los datos que se guardaran o actualizaran*

*\* @param***{array}***validParams*

*\* @param***{object}***body*

*\*/*

function **buildParams**(validParams, body) {

    let params = {};

    validParams.**forEach**(attr => {

        if (**Object**.prototype.**hasOwnProperty**.**call**(body, attr))

            params[attr] = body[attr];

    });

    return params;

}

module.exports = { **buildParams** };

En el controlador de places se modifica el actualizar y el crear

*// importar el controlador*

const helpers = **require**('./helpers');

validParams = [

    'title', 'description', 'address', 'acceptsCreditCard',

    'openHour', 'closeHour'

];

function **update**(req, res) {

*// actualizar un registro*

    const params = helpers.**buildParams**(validParams, req.body);

    req.place = **Object**.**assign**(req.place, params);

*// Object.assign(target , source)*

*// copia todos las propiedades de la fuente(source) al objetivo (target)*

    req.place.**save**().**then**(doc => {

        res.**json**(doc);

    }).**catch**(err => {

        console.**log**(err);

        res.**json**(err);

    });

}

function **create**(req, res, next) {

*// crear un sitio*

    const params = helpers.**buildParams**(validParams, req.body);

**Place**.**create**( params )

        .**then**(doc => {

*// res.json(doc)*

            req.place = doc;

**next**();

        }).**catch**(err => {

*// console.log(err);*

*// res.json(err);*

**next**(err);

        });

}

En el modelo Place se agrega **address** al esquema

## Módulo de usuarios

Primero creamos el modelo “User.js”

const **mongoose** = **require**('mongoose');

let userSchema = new **mongoose**.**Schema**({

    email: {

**type**: **String**,

        required: true,

        unique: true

    },

**name**: **String**,

    admin: {

**type**: **Boolean**,

        default: false

    }

});

const **User** = **mongoose**.**model**('User', userSchema);

module.**exports** = **User**;

Además, encriptaremos la contraseña para ello utilizamos bcrypt, instalamos la librería

npm install mongoose-bcrypt

const **mongoose** = **require**('mongoose');

const mongooseBcrypt = **require**('mongoose-bcrypt');

let userSchema = new **mongoose**.**Schema**({

    email: {

**type**: **String**,

        required: true,

        unique: true

    },

**name**: **String**,

    admin: {

**type**: **Boolean**,

        default: false

    }

});

*/\*\**

*\* agrega un campo encriptado llamado password (cuando guardemos un password este sera encriptado)*

*\* con unos metodos verifyPassword(password,callback), verifyPasswordSync(password)*

*\* y static method encryptPassword(password,callback)*

*\*/*

userSchema.**plugin**(mongooseBcrypt);

const **User** = **mongoose**.**model**('User', userSchema);

module.**exports** = **User**;

Importamos en app.js la ruta users, en caso de que no exista debe crearse

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

const placesRouter = **require**('./routes/places');

const usersRouter = **require**('./routes/users');

*// app.use('/', indexRouter);*

app.**use**('/places', placesRouter);

app.**use**('/users', usersRouter);

Creamos la ruta users, con la ruta para crear un usuario

const **express** = **require**('express');

const router = **express**.**Router**();

const usersController = **require**('../controllers/UsersController');

*/\* GET users listing. \*/*

router.**route**('/')

  .**post**(usersController.**create**)

module.exports = router;

Creamos el controlador Users con el método créate

const **User** = **require**('../models/User');

const **buildParams** = **require**('./helpers').**buildParams**;

const validParams = ['email', 'name', 'password'];

function **create**(req, res) {

    let params = **buildParams**(validParams, req.body);

**User**.**create**(params)

        .**then**(user => {

            res.**json**(user);

        }).**catch**(error => {

            console.**log**(error);

            res.**status**(422).**json**({

                error

            })

        })

}

module.exports = { **create** };

### Json web Token

Pasos

* Generar los tokens
* Enviar los tokens
* Validar los tokens

$ npm install jsonwebtoken

Creamos un controlador middleware para las sesiones “SessionController.js”

const jwt = **require**('jsonwebtoken');

const secrets = **require**('../config/secrets');

function **generateToken**(req, res, next) {

    if (!req.user) return **next**();

*// si existe req.user almacenamos dentro la propiedad token el token generado*

*// se puede almacenar varios datos, en este caso solo el id*

*// lo segundo es el secreto con lo que se firman los tokens*

    req.token = jwt.**sign**({ id: req.user.\_id }, secrets.jwtSecret);

**next**();

}

function **sendToken**(req, res) {

    if (req.user) {

        res.**json**({

            user: req.user,

            jwt: req.token

        })

    } else {

        res.**status**(422).**json**({

            error: 'Could not create user'

        })

    }

}

module.exports = { **generateToken**, **sendToken** }

Ajustamos el método créate del controlado UsersController, para que de paso a los siguientes métodos de la sesión

**User**.**create**(params)

        .**then**(user => {

*// res.json(user);*

            req.user = user;

**next**();

        })

En las rutas de users

Importamos el controlador de sesiones

const sessionController = **require**('../controllers/SessionsController');

Y agregamos los métodos de sesión a la ruta post

router.**route**('/')

  .**post**(

    usersController.**create**,

    sessionController.**generateToken**,

    sessionController.**sendToken**

  )

### Iniciar sesión

En el controlador de sesiones agregamos un método para autenticación

const **User** = **require**('../models/User');

function **authenticate**(req, res, next) {

**User**.**findOne**({ email: req.body.email })

        .**then**(user => {

            user.**verifyPassword**(req.body.password)

                .**then**(valid => {

                    if (valid) {

                        req.user = user;

**next**();

                    } else {

**next**(new **Error**('Invalid Credentials'));

                    }

                })

        }).**catch**(error => **next**(error));

}

Agregamos una nueva ruta “sessions”

En app.js

// se importa

const sessionsRouter = **require**('./routes/sessions');

// se usa

app.**use**('/sessions', sessionsRouter );

Creamos la ruta sessions

const **express** = **require**('express');

const router = **express**.**Router**();

const sessionsController = **require**('../controllers/SessionsController');

router.**route**('/')

    .**post**(

        sessionsController.**authenticate**,

        sessionsController.**generateToken**,

        sessionsController.**sendToken**

    )

module.exports = router;

### Que solo el primer usuario sea admin

En el modelo User.js

*// hook que guarda el primer usuario como administrador*

userSchema.**post**('save', function(user, **next**) {

**User**.**count**({}).**then**(count => {

        if (count == 1) {

**User**.**updateOne**({ '\_id': user.\_id }, { admin: true })

                .**then**(result => {

**next**();

                })

        } else {

**next**();

        }

    })

})

Para eliminar todos los registros en el controlador de usuarios

*// function destroyAll(req, res) {*

*//     User.remove({}).then(r => res.json({}));*

*// }*

Y en la ruta de usuarios

*// .get(usersController.destroyAll);*