

Refuerzo Backend inicio de sesión

Objetivos del momento:

Crear un nuevo proyecto en node.js, express, mysql con sequelize-cli.

Instalación y configuración de paquetes.

Creación de api para la integración con el frontend.

ejecución de pruebas unitarias con marco de pruebas jest (ajustado a una sesión posterior)

Crear un nuevo proyecto

Cree un directorio para que contenga la aplicación y conviértalo en el directorio de trabajo.

ejecutamos los siguientes comandos desde una terminal, ubicados en la carpeta donde alojamos nuestros proyectos:

```
mkdir Nombre-carpeta-proyecto
```

```
cd Nombre-carpeta-proyecto
```

ejemplos:

```
$ mkdir myapp
```

```
$ cd myapp
```

```
C:\Users\andre\Desktop>mkdir backend-login
```

```
C:\Users\andre\Desktop>cd backend-login
```

e inicializamos un nuevo proyecto con el comando npm init:

```
$ npm init
```

```
C:\Users\andre\Desktop\prueba-sequelize-cli>npm init
```

Instalación y configuración de paquetes.

A continuación, se realiza la instalación de los paquetes que vamos a requerir para el desarrollo del proyecto.

Para la instalación de los diferentes paquetes, se utilizará el comando npm install de la siguiente manera:

```
$ npm install cors jsonwebtoken bcryptjs --save
```

```
C:\Users\andre\backend_login>npm install _cors jsonwebtoken bcryptjs --save
```

y ahora instalamos el segundo grupo de paquetes, nuevamente con npm install:

```
$ npm install --save express body-parser sequelize sequelize-cli  
sqlite3 mysql2 nodemon morgan
```

cors : Este es un middleware que se puede usar para habilitar CORS con varias opciones. Básicamente, permitir peticiones del mismo origen para nuestro caso.

bcrypt : Esto nos ayudará a codificar las contraseñas de los usuarios antes de almacenarlas en la base de datos.

jsonwebtoken: JSON Web Token (JWT) se utilizará para la autenticación y autorización. Este paquete ayudará a configurar rutas protegidas a las que solo pueden acceder los usuarios registrados.

sequelize-cli :ayuda a automatizar algunas de las partes mundanas y no triviales de la programación de bases de datos.

nodemon: Nos servirá para contar con un live server y no reiniciar nuestro servidor cada que hagamos un cambio

sqlite3 y mysql2: son los drivers necesarios para funcionar con las bases de datos.

body-parser: para trabajar con mensajes tipo JSON, es opcional ya que express permite dicha funcionalidad.

morgan: Permite registrar las peticiones http y ver el log de algunos datos, es opcional.

Iniciando configuraciones

Iniciamos un nuevo proyecto de sequelize con el comando: `npx sequelize-cli init`

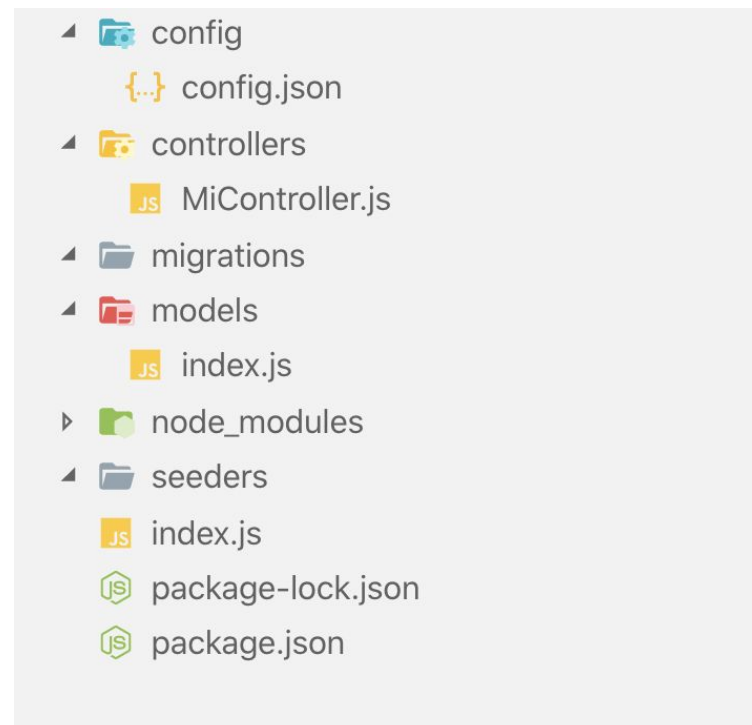
```
$ npx sequelize-cli init
```

```
(base) DdMAC:Back-sqlzcli usuarioutp$ npx sequelize-cli init
Sequelize CLI [Node: 10.10.0, CLI: 6.2.0, ORM: 6.3.5]

Created "config/config.json"
Successfully created models folder at "/Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/models".
Successfully created migrations folder at "/Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/migrations".
Successfully created seeders folder at "/Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/seeders".
```

Se utiliza npx debido a que sequelize **no** se ha instalado de forma global y no se reconoce como un comando del sistema.

Configure y actualice su proyecto hasta obtener la siguiente estructura:



Solo ha sido necesario crear la carpeta controllers con su respectivo controlador.js y el index.js principal.

Creación de api para la integración con el frontend.

SQLITE

En primer lugar, necesitamos editar el archivo config / config.json para que sequelize sepa que vamos a trabajar con una base de datos SQLite. borramos toda la información que se encuentra allí y copiamos el siguiente fragmento de código:

```
{
  "development": {
    "dialect": "sqlite",
    "storage": "./database.sqlite3"
  },
  "test": {
    "dialect": "sqlite",
    "storage": "./database.sqlite3"
  },
  "production": {
    "dialect": "sqlite",
    "storage": "./database.sqlite3"
  }
}
```

que creará y utilizará un archivo de base de datos SQLite llamado database.sqlite3 en la raíz del proyecto.

MYSQL

En caso de querer hacerlo utilizando mysql solo necesitamos cambiar un poco la configuración:

```
{
  "development": {
    "username": "yourDataBaseUserName",
    "password": "yourDataBasePassword",
```

```
"database": "yourDataBaseName",
"host": "yourDataBaseHost",
"dialect": "mysql"
},
"test": {
  "dialect": "sqlite",
  "storage": "./database.sqlite3"
},
"production": {
  "dialect": "sqlite",
  "storage": "./database.sqlite3"
}
}
```

Ahora seguimos con otro comando, pero esta vez usaremos el `model generate` argumento para definir mi modelo de user y sus atributos, de la siguiente manera

- Con instalación global de sequelize:

```
$ sequelize model:generate --name user --attributes
name:string,password:string,email:string
```

```
sequelize model:generate --name user --attributes name:string,password:string,email:string
```

o con npx

```
npx sequelize-cli model:generate --name user --attributes
name:string,password:string,email:string
```

```
(base) DdMAC:Back-sqlzcli usuarioutp$ npx sequelize-cli model:generate --name user --attributes name:string,password:string,email:string
```

```
Sequelize CLI [Node: 10.10.0, CLI: 6.2.0, ORM: 6.3.5]
```

```
New model was created at /Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/models/user.js .
New migration was created at /Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/migrations/20201211040745-create-user.js .
(base) DdMAC:Back-sqlzcli usuarioutp$
```

El parámetro `--name` es obviamente el nombre del modelo a generar y el parámetro `--attributes` va seguido de los campos de objeto que lo definen junto con sus tipos de datos. Las salidas de este comando son dos archivos nuevos:

En la carpeta `models` un nuevo modelo llamado `user.js`: un modelo de datos que se utilizará en el código lógico de la aplicación `Node.js`



```
'use strict';
const {
  Model
} = require('sequelize');
module.exports = (sequelize, DataTypes) => {
  class user extends Model {
    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate(models) {
      // define association here
    }
  };
  user.init({
    name: DataTypes.STRING,
    password: DataTypes.STRING,
    email: DataTypes.STRING
  }, {
    sequelize,
    modelName: 'user',
  });
  return user;
};
```

migrations / aaaammddHHMMSS-create-user.js: un script de migración que emitirá DDL SQL para crear la tabla de contactos en la base de datos.

```
'use strict';
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      name: {
        type: Sequelize.STRING
      },
      password: {
        type: Sequelize.STRING
      },
      email: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('users');
  }
};
```

Además de los atributos especificados en el model:generate comando sequelize-cli también generará un número entero auto-incrementales id campo, así como como createdAt y updatedAt campos de fecha de enlace.

Migración de DB

Lo siguiente que debe hacer es ejecutar la migración para que la base de datos SQLite o mysql contenga la tabla de contactos de la siguiente manera:

```
C:\Users\andre\Desktop\backend-login>sequelize db:migrate
```

O

```
npx sequelize-cli db:migrate
```

```
...
(base) DdMAC:Back-sqlzcli usuarioutp$ npx sequelize-cli db:migrate
```

```
Sequelize CLI [Node: 10.10.0, CLI: 6.2.0, ORM: 6.3.5]
```

```
Loaded configuration file "config/config.json".
```

```
Using environment "development".
```

```
== 20201211040745-create-user: migrating =====
```

```
== 20201211040745-create-user: migrated (1.007s)
```

Este comando indicará que la migración se ha ejecutado correctamente. Ahora puedo abrir mi archivo database.sqlite3 recién generado y ver el esquema.

Para abrir el archivo ejecuto el siguiente comando:

```
$ sqlite3 database.sqlite3
```

luego en la terminal se debe llamar a .schema

```
sqlite> .schema
```

ahora puedo visualizar mi schema en sqlite


```
CREATE TABLE `SequelizeMeta` (`name` VARCHAR(255) NOT NULL UNIQUE  
PRIMARY KEY);  
CREATE TABLE `users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT,  
`name` VARCHAR(255), `password` VARCHAR(255), `email`  
VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME  
NOT NULL);  
CREATE TABLE sqlite_sequence(name,seq);
```

para salir de la consola de sqlite debe usar:

```
sqlite> .exit
```

para MYSQL podemos revisar la respuesta por consola o consultar la base de datos para revisar el esquema de la tabla creada.

Para finalizar con el cli de sequelize, solo necesitamos crear un registro de prueba con el que podamos ejecutar los módulos que desarrollamos durante las clases sincrónicas.

para poner en marcha nuestros módulos tenemos que crear un usuario el cual vamos a autenticar, para esto generamos un usuario por medio de seeders, para probar nuestra aplicación de inicio de sesión para esto ejecutamos el siguiente comando:

```
sequelize seed:generate --name seed-user
```

O

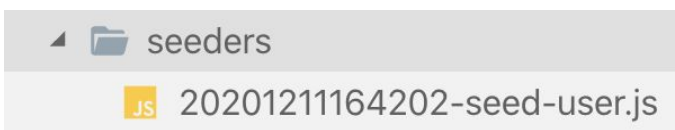
```
npx sequelize-cli seed:generate --name seed-user
```

La consola nos debe entregar una respuesta como la siguiente:

```
(base) DdMAC:Back-sqlzcli usuarioutp$ npx sequelize-cli seed:generate --name seed-user
```

```
Sequelize CLI [Node: 10.10.0, CLI: 6.2.0, ORM: 6.3.5]
```

```
seeders folder at "/Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/seeders" already exists.  
New seed was created at /Users/usuarioutp/Downloads/BackCode/Back-sqlzcli/seeders/20201211043541-seed-user.js .
```



El resultado es un nuevo script en el directorio de seeders de la convención de nomenclatura aaaammddHHMMSS-seed-user.js. lo actualizamos con el siguiente fragmento de código:

```
'use strict';

module.exports = {

  up: async(queryInterface, Sequelize) => {

    return queryInterface.bulkInsert('Users', [{

      name: 'carlos',

      email: 'ejemplo@gmail.com',

      password: '',

      createdAt: new Date(),

      updatedAt: new Date()

    }]);

  },

  down: async(queryInterface, Sequelize) => {

    return queryInterface.bulkDelete('Users', null, {});

  }

};
```

Pero esta vez el campo contraseña lo encriptamos en la siguiente plataforma web:
<https://bcrypt-generator.com/> de esta forma

Encrypt

Encrypt some text. The result shown will be a Bcrypt encrypted hash.

String to encrypt	Hash!
8	Rounds

Presionamos el botón hash y éste nos generará el texto encriptado y le damos copiar, lo ingresamos en campo contraseña de nuestra bd tabla user que estamos creando.

Por último, necesitamos ejecutar el seeder para completar la base de datos con el user de prueba.

```
sequelize db:seed:all
```

o

```
npx sequelize-cli db:seed:all
```

Lo que me da un resultado en la consola que me permite saber en la tabla de la base de datos se creo el usuario correctamente con datos

```
(base) DdMAC:Back-sqlzcli usuarioutp$ npx sequelize-cli db:seed:all
```

```
Sequelize CLI [Node: 10.10.0, CLI: 6.2.0, ORM: 6.3.5]
```

```
Loaded configuration file "config/config.json".
```

```
Using environment "development".
```

```
== 20201211043541-seed-user: migrating =====
```

```
== 20201211043541-seed-user: migrated (0.208s)
```

Nota: A partir de este punto, puede aplicar las rutas, controladores y demás configuraciones vistas durante las sesiones de clase, sin embargo, se deja un código adicional por si desea revisarlo:

Código Adicional

Luego actualizamos el archivo server.js como se observa a continuación:

server.js

```
const controller = require('./controller/controller.js');
const express = require('express');
const db = require('./models');
const app = express()
const bodyParser = require('body-parser');
app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
    next();
});

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }));

// API ENDPOINTS
app.get('/api/users', (req, res) => {
    db.user.findAll().then(users => res.json(users))
});
app.post('/api/auth/signin', controller.signin);

app.get('/', function(req, res) {
    db.user.findAll().then(users => res.json(users))
});
const port = 3000
app.listen(port, () => {
    console.log(`Running on http://localhost:${port}`)
})

module.exports = app;
```

Este fragmento de código, sin embargo, habilitaría CORS para todos los recursos en su servidor.

```
app.use(function(req, res, next) {  
  res.header("Access-Control-Allow-Origin", "*");  
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");  
  next();  
});
```

Agregamos una nueva ruta, la cual será consumida desde frontend que se desarrolló en la sesión anterior :

```
app.post('/api/auth/signin', controller.signin);
```

como observamos la ruta por medio del método post ya que si recordamos desde el frontend utilizamos un formulario que nos envía los datos de autenticación del usuario email y contraseña, tal información se la enviamos al controlador que lo tenemos ubicado en el directorio controller/controller.js y no nos olvidemos de importarlo en el archivo index.js como se observó anteriormente.

```
const controller = require('./controller/controller.js');
```

en el archivo controller.js, copiamos el siguiente código:

```
const config = require('../secret/config.js');  
const db = require('../models');  
var jwt = require('jsonwebtoken');  
var bcrypt = require('bcryptjs');  
  
exports.signin = (req, res) => {  
  db.user.findOne({  
    where: {  
      email: req.body.email  
    }  
  
  }).then(user => {  
    if (!user) {  
      return res.status(404).send('User Not Found.');    }  
  });  
}
```

```

    }

    var passwordIsValid = bcrypt.compareSync(req.body.password,
user.password);
    if (!passwordIsValid) {
        return res.status(401).send({ auth: false, accessToken: null,
reason: "Invalid Password!" });
    }

    var token = jwt.sign({ id: user.id, name: user.name, email:
user.email }, config.secret, {
        expiresIn: 86400 // expires in 24 hours
    });

    res.status(200).send({ auth: true, accessToken: token });

}).catch(err => {
    res.status(500).send('Error -> ' + err);
});
}

```

El archivo tiene una función llamada **signin** básicamente lo que hace es recibir los datos que fueron ingresados en el formulario en este caso email y contraseña, y realizar las validaciones,

Lo primero validamos si el usuario existe en la bd por medio del email usando sequelize para realizar la consulta:

```

db.user.findOne({
  where: {
    email: req.body.email
  }

}).then(user => {
  if (!user) {
    return res.status(404).send('User Not Found.');
```

Si el usuario no existe retornamos un estado 404 diciendo que usuario no se encuentra, si de lo contrario el usuario existe, sigue la siguiente validación que es verificar si la contraseña ingresada es igual a la contraseña almacenada en la bd, por seguridad del usuario tal contraseña debe estar encriptada en la bd para poder comparar tal contraseña almacenada utilizamos el paquete 'bcryptjs', de la siguiente manera:

Importación del paquete:

```
var bcrypt = require('bcryptjs');
```

validación de contraseña:

```
var passwordIsValid = bcrypt.compareSync(req.body.password, user.password);
;
    if (!passwordIsValid) {
        return res.status(401).send({ auth: false, accessToken: null, reason: "Invalid Password!" });
    }
```

Podemos observar que si la contraseña es incorrecta respondemos con un estado 401 de Unauthorized, token=null.

De lo contrario, Primero creamos un token de autenticación para enviarlo al frontend, para esto hacemos uso del paquete jwt

Importamos el paquete:

```
var jwt = require('jsonwebtoken');
```

Creamos token:

```
var jwt = require('jsonwebtoken');
var token = jwt.sign({ id: user.id, name: user.name, email: user.email }, config.secret, {
    expiresIn: 86400 // expires in 24 hours
});
```

Observamos que hacemos uso de jwt.sign donde le pasamos varios parámetros información del usuario, una llave secreta que se encuentra en config.js

Con la siguiente información:

```
module.exports = {
    'secret': 'key-super-secret',
};
```

¿Y qué significa?, bueno, básicamente JWT necesita una “contraseña maestra” por así llamarlo que usará para encriptar la información y aquí la pondremos, ustedes deberán usar una clave segura que nadie pueda conocer fácilmente.

Si todo sale bien responde con un estado 200 y con el token recién creado que nos permitirá autenticar al usuario en nuestro frontend.

```
res.status(200).send({ auth: true, accessToken: token });
```

integración backend y frontend

básicamente la integración del módulo de inicio de sesión consiste desde front end consumir la ruta creada anteriormente <http://localhost:3000/api/auth/signin> por medio de una solicitud axios que realizamos en la sesión anterior:

src/components/auth/login.vue (frontend)

```
let response = await this.$http.post("/api/auth/signin", this.login);
```

El siguiente paso es levantar los servidores tanto del backend como del frontend de la siguiente manera:

Backend:

```
PROBLEMS  TERMINAL  OUTPUT  ...  1: cmd  +  -  x
Microsoft Windows [Versión 10.0.18363.1198]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

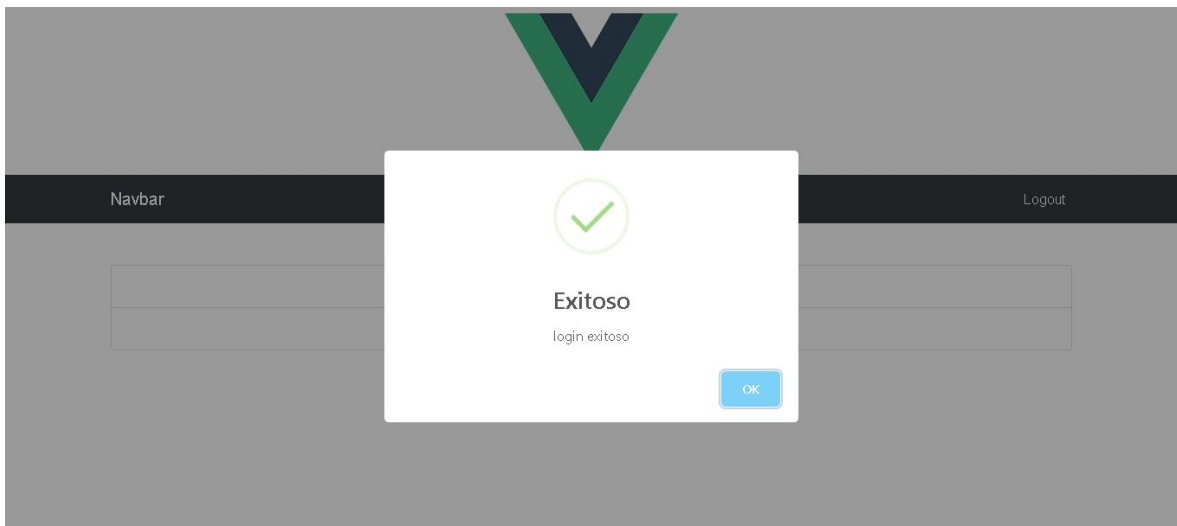
C:\Users\andre\Desktop\back-login>nodemon index.js
```

Frontend:

```
PROBLEMS  TERMINAL  OUTPUT  ...  1: cmd  +  -  x
Microsoft Windows [Versión 10.0.18363.1198]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\andre\Desktop\proyecto_login\proyecto_login_frontend>npm run serve
```

Al completar el proceso al ejecutar los comando ,copiamos la siguiente ruta en nuestro navegador <http://localhost:8081/>. ingresamos la credenciales de email y contraseña(sin encriptar ejemplo:micontraseña) del usuario creado anteriormente en la db.



Name : carlos
Email : carlos24lg@gmail.com

Podemos observar que el usuario queda autenticado y así podremos ingresar a la ruta home que tiene tal restricción de autenticación.