MVC

Denis Pacheco Bollmann - 2022

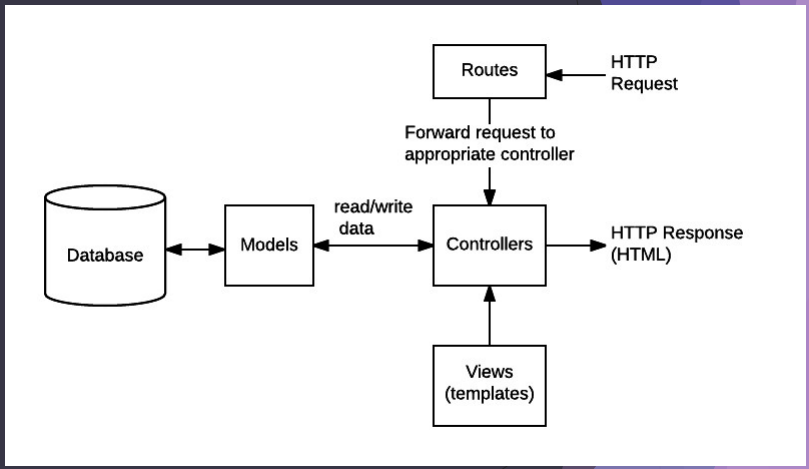
pachinx@gmail.com

Qué es MVC?

- MVC viene del acrónimo Model View Controller
- Es una qruitectura de diseño por medio de la cual se separa la funcionalidad del sistema en tres capas:
- Model: todo el código que tiene que ver con la base de datos. Se encarga de generar la estructura de los modelos/tablas y consultas
- View: lo que ve el usuario/cliente: html y css
- Controller: es el encargado de la lógica, revisa los request y genera la data que será enviada al usuario/cliente.

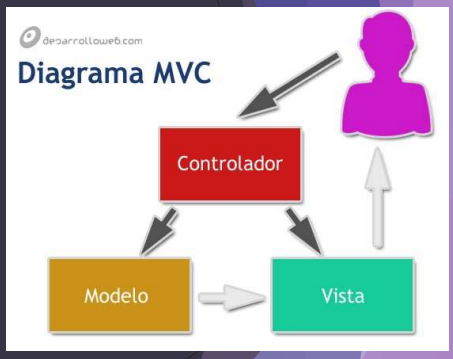
MVC

- Observamos que el punto de entrada son las rutas
- El punto de salida es el controlador (que se alimenta de los datos del modelo y el visualización de la vista)
- El modelo es el que se conecta a la BD



Por qué MVC?

- Para separar tecnologías: sql para el modelo, html/css para la vista, js para el controlador.
- Distintos desarrolladores pueden trabajar en distintas partes del proyecto.
- Diseño modular.
- Rapidez, Reutilización de código, mantención.



Implementando MVC en Node.js

- Para implementar un proyecto MVC en node, necesitamos trabajar con módulos que nos permitan este trabajo.
- Para este caso, las principales son:
- Sequelize: nos permite trabajar con modelos
- Ejs: nos permite trabajar con vistas
- Express: nos permite trabajar con rutas
- Además de estas, necesitaremos trabajar con unas cuantas mas.

Creando un nuevo proyecto

- Vamos empezar creando un nuevo proyecto.
- Como paso previo, podemos generar un nuevo repositorio en Github.
- Una ves creada la carpeta, la abrimos con visual studio code y generamos el proyeto:
- $ npm init -y
- Ahora, instalaremos las librerías o módulos necesarios:

```
$ npm install express nodemon sequelize sequelize-cli moment dotenv ejs express-fileupload chalk@4.1.0
```

```
npm install pg pg-store
```

Configuración Proyecto MVC

- El archivo package.json deberá quedar así:

```
{
  "dependencies": {
    "chalk": "^4.1.0",
    "dotenv": "^16.0.1",
    "ejs": "^3.1.8",
    "express": "^4.18.1",
    "express-fileupload": "^1.4.0",
    "moment": "^2.29.4",
    "nodemon": "^2.0.19",
    "pg": "^8.7.3",
    "pg-hstore": "^2.3.4",
    "sequelize": "^6.21.3",
    "sequelize-cli": "^6.4.1"
  }
}
```

- Además, agregaremos un nuevo archivo llamado nodemon.json (en la raíz de nuestro proyecto) que contendrá lo siguiente:

{} nodemon.json > ...

```
1  {
2      "ext": "js,json,hbs,ejs"
3  }
```

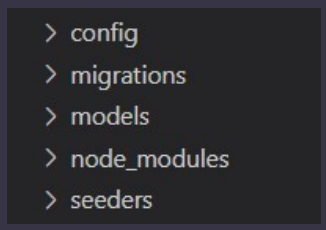
- Esto para que nodemon detecte los cambios en nuestras vistas

Inicializando sequelize

- Lo primero que vamos a hacer, es inicializar sequelize para poder trabajar con modelos y migraciones.
- Esto lo haremos para abstraernos del código SQL y sólo trabajar con funciones.

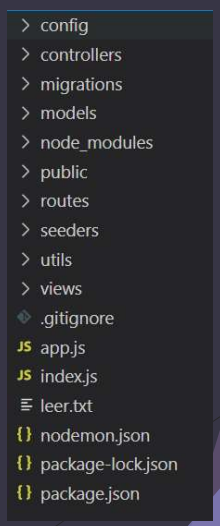
```
$ npx sequelize-cli init
```

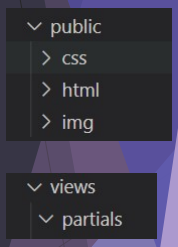
- Como sabemos, esto nos creará carpetas de trabajo para los modelos:



Creando estructura de trabajo MVC

- Ahora agregaremos las siguientes carpetas a nuestro proyecto:
- views: para las vistas ejs
- views/partials: para los partials de las vistas
- controllers: para los controladores
- routes: rutas de express
- public: para archivos de acceso publico
- public/html: archivos html de acceso público
- public/css: archivos css de acceso público
- public/js: archivos js de acceso público
- utils: para funciones comunes
- Archivo app.js: para cargar aplicación
- Archivo index.js: para ejecutar aplicación
- Archivo .env: para credenciales
- Archivo .gitignore: para omitir carpetas y archivos en git





Cómo comenzar?

- Lo más fácil es empezar por el principio :p
- Vamos a generar un módulo para la aplicación de express en nuestro archivo app.js, la exportaremos con el nombre app
- En index.js importaremos esta app y la iniciaremos, para hacer andar el server.
- Por último, lo ejecutaremos con nodemon:
- Podemos escribir sólo nodemon en la línea de comandos, ya que por defecto nos toma el index como archivo de inicio.

Comenzando nuestra aplicación

- 1.- iniciamos la app de express en app.js:

app.js > ...

```
1   //carga de librerias---------------------------------
2   const express=require("express");
3   //configuraciones-----------------------------------
4   //express
5   app=express();
6   app.use(express.static(__dirname+"/public"));
7   app.set("view engine","ejs");
8   app.set("views",__dirname+"/views");
9
10  module.exports=app;
```

Comenzando nuestra aplicación

- 2.- Creamos una variable de entorno para el puerto de escucha:

.env

```
1   APP_PORT=3000
2
```

- 3.- iniciamos el server en index.js:

index.js > ...

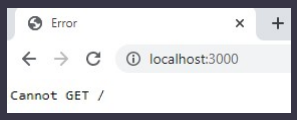
```
1   //carga de librerías---------------------------------
2   const chalk=require("chalk");
3   const app=require("./app");
4   require("dotenv").config();
5   //inicio del server----------------------------------
6   app.listen(process.env.APP_PORT, function() {
7       console.log("**********"+chalk.cyan.inverse("servidor iniciado en el puerto "+process.env.APP_PORT)+"**********");
8   })
```

Comenzando nuestra aplicación

- Ejecutamos:

```
XP@XP-PC-11 MINGW64 /x/Trabajo/Relatorias/FullStack con Node.js/Módulo 8 - Express/codigo/jwt_test
$ nodemon
[nodemon] 2.0.18
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,json,hbs,ejs
[nodemon] starting `node index.js`
**********servidor iniciado en el puerto 3000***********
```

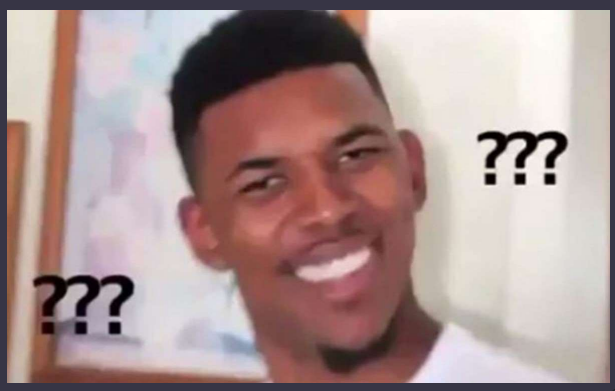
- Probamos:



- Funciona! Obviamente no nos va a mostrar anda porque no tenemos ninguna ruta/vista configurada ;)

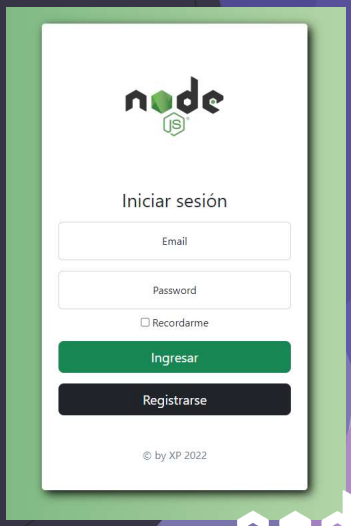
FIN

- Mentira, con esto sólo tenemos el servidor andando. Vamos por parte dijo Jack el destripador.



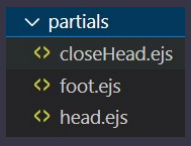
Desarrollar la vista

- Para esta clase, la idea es que preparemos el proyecto para crear una aplicación que nos permita entrar sólo si iniciamos sesión/registramos.
- Por lo tanto la primera vista que crearemos será una ventana de login.
- Para esto pasé semanas trabajando en una.
- Mentira, la copié de los ejemplos de bootstrap
- Por lo tanto, necesitaremos bootstrap y css, por lo que crearemos las vistas y partials necesarios.

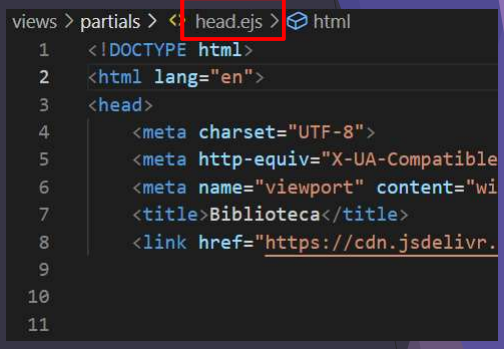


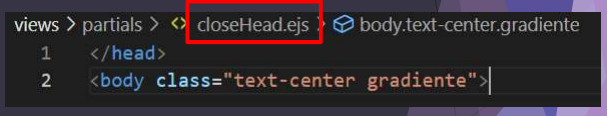
Partials

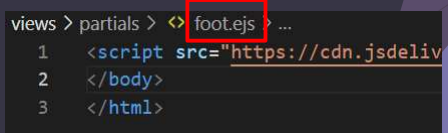
- Crearemos tres:



- head: para todo el encabezado, pero sin cerrarlo
- closeHead: sólo para cerrar el encabezado
- foot: para cerrar el body
- Agregar la carga de bootstrap en el head (css) y foot (js).
- Al body le agregué dos clases para que quede mas cachilupi. estas serán globales, una para centrar el contenido y la otra para un fondo de color.



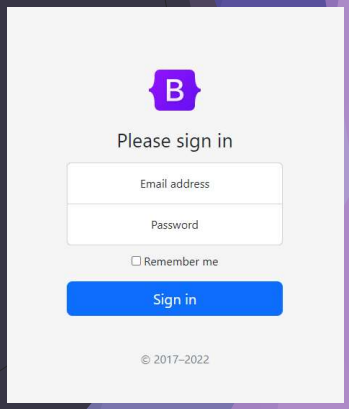




Vistas

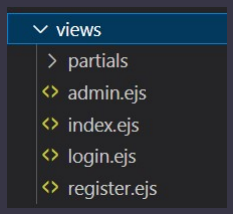
- Agregamos el archivo login.ejs en la carpeta views:
- La vista la pueden copiar de aquí:
- https://getbootstrap.com/docs/5.2/examples/sign-in/?
- Sólo abran la consola y copien el html (sólo la etiqueta main y su contenido)
- Habrá que hacerle algunas correcciones, ya que nos faltará algunos elementos
- Pero no sabremos qué hasta que podamos cargarla en una ruta!

```
<!DOCTYPE html>
<html lang="en">
▶ <head>…</head>
▼ <body class="text-center">
  ▼ <main class="form-signin w-100 m-auto"> == $0
    ▶ <form>…</form>
    </main>
  </body>
</html>
```



Vistas

- Agregar las vistas de register.ejs, index.js y admin.js, por ahora sólo con un texto.
- No olviden agregar los imports de los partials en cada una, incluyendo login.ejs.



views > index.ejs > ? > ? > h1

```
1  <%- include ("./partials/head")    %>
2  <%- include ("./partials/Closehead")    %>
3  <h1>INDEX</h1>
4  <%- include  ("./partials/foot")%>
```

views > register.ejs > ? > ? > ?

```
1  <%- include ("./partials/head")    %>
2  <%- include ("./partials/Closehead")    %>
3  <h1>REGISTER</h1>
4  <%- include  ("./partials/foot")%>
```

Controlador, Ruta y Modelo

- Ya tenemos listas nuestras rutas, pero cómo verlas?
- Tendremos que crear la vista y el controlador para login.
- Cada elemento debe tener una vista, una ruta, un controlador y posiblemente un modelo.
- Para el login tenemos la vista, crearemos el controlador y la ruta, pero no necesitamos modelo, ya que no se "guarda" en la bd. Posiblemente podríamos crear un registro de sesión, donde guardemos la fecha y hora cada ves que el usuario ingrese. Pero eso mas adelante (?)
- Por ahora sólo nos concentraremos en visualizar la vista.

Primero el Controlador

- Porqué primero? Porque la ruta necesita usarlo, por lo tanto tiene que estar disponible primero.
- En el controlador generaremos la lógica para una petición, ya sea get, post, put o delete
- En este caso necesitamos dos funcionalidades:
- get login : mostrará la vista de login
- post login: iniciará sesión (si es que los datos son correctos)
- Por ahora implementaremos sólo la primera, ya que para la segunda nos falta JWT, que veremos mas adelante.

Controlador Login

- Lo primero que debemos hacer es crear un archivo login.js en la carpeta controllers
- Ahora, crearemos las funciones para cada ruta (get y post), las cuales deben tener la misma estructura que los callbacks de express, o sea:

```
function (req,res) {
…código
res.send,res.json, res.render, etc…
}
```

Controlador Login

- Primero el get:

```
controllers > JS login.js > <unknown>
1   const getLogin=function (req,res){
2       res.render("login");
3   }
```

- Esto sólo enviará la página(vista) de login al cliente
- Ahora el post:

```
5   const postLogin=function (req,res){
6       //esto lo vamos a completar despues
7       res.send("OK!");
8   }
```

- Sólo un mensaje.
- Por último, exportamos las funciones:

```
10  module.exports={getLogin,postLogin}
```

Ahora nuestras funciones las podemos usar simplemente importando el módulo
Izzi pizzi no?

Ruta login

- Ahora crearemos la ruta para login. Para esto añadimos un archivo llamado login.js en la carpeta routes
- Aquí lo que haremos es simplemente declarar las rutas, importar el controlador y usar las funciones del controlador donde corresponda:
- get->getLogin
- post->postLogin

Creando la ruta login

- 1.- importamos express y creamos un router, no una app. Esto porque el server no debe ser llamado aquí (está en la app.js y ejecutado en el index.js)
- Estas son simples rutas que agregaremos a esa app, en el archivo app.js

routes > JS login.js > ...

```
1  const express=require("express");
2  const router=express.Router();
```

- 2.- importamos el controlador (sus funciones) de login

```
//importar las funciones de su controlador correspondiente
const {getLogin,postLogin}=require("../controllers/login");
```

Creando la ruta login

- 3.- usando las funciones para cada ruta correspondiente:

```
6   router.get("/",getLogin);
7   router.post("/",postLogin);
```

- 4.- exportamos el router:

```
9   module.exports=router;
```

- En este momento se preguntarán: porqué la ruta es "/" y no "/login"?
- Por lo siguiente: en ves de llamar uno a uno los router (login, register, admin, etc.), los cargaremos todos juntos, usando el nombre del archivo. Con esto concatenaremos el "/" con el "login" de forma dinámica.
- Para esto es importante que el nombre del archivo sea el mismo de la ruta que se requiere implementar

Cargando rutas de forma dinámica

- Para poder usar estas rutas, es necesario llamarlas en la app de express (creada en app.js) con la función use. Por ejemplo:
- app.use ("/login",requiere("./routes/login"))
- app.use("/registro",require("./routes/registro"))
- app.use("/contacto",require("./routes/contacto"))
- Etc..
- Si fueran 100, tendríamos que cargar las 100 rutas!!
- Para evitar eso, las cargaremos de forma dinámica, a todas juntas y luego las cargaremos en una línea en nuestra app.

Cargando rutas de forma dinámica

- Para lograr esto, debemos crear una función muy similar a la de sequelize para cargar los modelos (en el archivo models/index.js)
- De hecho, podemos reutilizar parte del código.
- Esta carga la vamos a hacer en un nuevo archivo llamado index.js dentro de la carpeta routes.
- Este archivo lo llamaremos por defecto en la app mas adelante.
- Cabe señalar que este archivo no lo tocaremos nunca mas.
- Lo pueden guardar y reutilizar en futuros proyectos.

Cargando rutas de forma dinámica

routes > JS index.js > ...

```
1  //carga de librerias
2  const express=require("express");
3  const fs=require("fs");
4  const path = require('path');
5  //condfiguraciones
6  const router=express.Router();
7  //obtener el nombre de ESTE archivo (index.js)
8  const basename = path.basename(__filename);
9
10 fs
11   .readdirSync(__dirname) // busar todos los nombre de archivo de la carpeta
12   .filter(file => {
13     //filtrar para que no me tome según las condiciones:
14     //no considerar index.js (basename)
15     //que tengan extension .js
16     //que el punto no esté al principio (como en .env)
17     return (file.indexOf('.') !== 0) && (file !== basename) && (file.slice(-3) === '.js');
18   })
19   .forEach(file => {
20     //agregar cada ruta al router
21     router.use("/"+path.parse(file).name,require('./'+path.parse(file).base));
22   });
23
24   //exortar el router, este es el que usará la app
25   module.exports = router;
```

Y ahora?

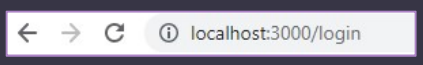
- Ya estamos listo. Sólo nos falta indicarle a express que use este archivo y cargue este router.
- Para esto nos vamos al archivo app.js y agregamos lo siguiente:

```
//carga de librerias--------------------
const express=require("express");
const router=require("./routes");
```

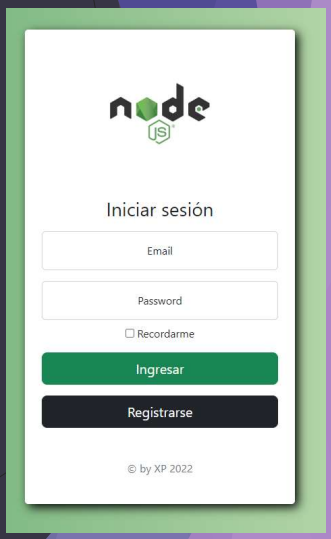
```
//express
app=express();
app.set("view engine","ejs");
app.set("views",__dirname+"/views");
app.use(express.static(__dirname+"/public"));
app.use("/",router);
```

Hora de testing!

- Si teníamos nodemon andando todo este tiempo, pues no necesitamos hacer nada mas.
- De lo contrario, deben iniciarlo.
- La única ruta que tenemos creada es "login", por lo que tendremos que visitar la url:



- Verán la imagen de la derecha…. O no??
- Seguro tiene algunos problemas, los corregiremos enseguida.



Algunas correcciones: Logo

- 1.- Seguramente la imagen no se ve, busquen una y guárdenla en la carpeta public/img con el nombre de logo.png o la extensión que corresponda.
- 2.- Crear un archivo llamado login.css en la carpeta public/css
- 3.- dar formato a la imagen:
- Crear un estilo(clase) en el archivo login.css llamado "logo" y asignárselo a la imagen en la vista login.ejs, junto con la ruta y nombre de la nueva imagen.

```
.logo{
    width: 200px;
}
```

```
8        <img class="mb-4 logo" src="img/logo.png" alt="">
```

Cargar el css

- Para cargar el css, debemos agregar la línea entre los partials head y closeHead en el archivo login.ejs

views > login.ejs > ? > ? > main.form-signin.w-100.m-auto.f

```
1   <%- include ("./partials/head")    %>
2   <link rel="stylesheet" href="css/login.css">
3   <%- include ("./partials/Closehead")    %>
```

Algunas correcciones: Formato

- Para que el formulario tenga un tamaño correcto, necesitamos crear el estilo (clase) “form-signin”.

```
.form-signin {
    max-width: 400px;
    padding: 25px;
}
```

- Esta clase ya viene asignada, así que no debemos agregar nada, sólo que no era parte de bootstrap (si estaba en la página donde la copiamos)

Algunas correcciones: Formulario

- Ahora agregaremos un nuevo estilo (clase), este formato lo agregamos según nuestro gusto. Es para darle un color de fondo al formulario y un borde mas atractivo:

```
.formulario{
    background-color: white;
    border: 1px black;
    border-radius: 5px;
    box-shadow: 5px 5px 15px black;
}
```

- Ahora debemos agregarlo al form:

```
6    <main class="form-signin w-100 m-auto formulario mt-5">
```

Algunas correcciones: Fondo y centrado

- Para agregar el fondo (opcional), simplemente creamos el estilo y se lo damos al body:

```
.gradiente{
    /* navegadores antiguos */
    background: #029009;

    /* Chrome 10-25, Safari 5.1-6 */
    background: -webkit-linear-gradient(to right, rgb(0, 147, 42), rgb(175, 251, 156));

    /* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */
    background: linear-gradient(to right, rgb(58, 149, 84), rgb(248, 249, 213) )
    }
```

- En el archivo closeHead.ejs agregamos el estilo y el centrado (de bootstrap):

views > partials > closeHead.ejs > body.text-center.gradiente

```
1   </head>
2   <body class="text-center gradiente">
```

Ultimos ajustes

- Podemos jugar con el espaciado entre los elementos, usando algún margen de bootstrap.
- Por ejemplo, para que el email y password no estén tan juntos, agregamos el margen mb-3 al email:

```
12          <input type="email" class="form-control mb-3"
```

- También al password un mb-2, para despegarlo del “recordarme”:

```
16          <input type="password" class="form-control mb-2"
```

Otros Ajustes

- Podemos agregar también otro botón, para el registro:

```
24          </div>
25          <button class="w-100 btn btn-lg btn-success mb-3" type="submit">Ingresar</button>
26          <button class="w-100 btn btn-lg btn-dark" type="submit">Registrarse</button>
27          <p class="mt-5 mb-3 text-muted">© by XP 2022</p>
28      </form>
```

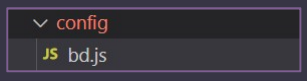
- Y cambiar textos y colores.
- Ya estamos listos con la vista, las rutas y el controlador GET del login.
- Ahora implementaremos el controlador POST

Modelo

- Antes de implementar el login, debemos tener los registros en la BD.
- Para esto usaremos la línea de comandos (CLI) de sequelize
- Los modelos son los siguientes:
- Tabla Usuarios: email(PK), username(único), nombre, password, idRol(FK)
- Tabla Roles: idRol(PK), Nombre

Configuración

- Lo primero que haremos es configurar la conexión a la base de datos. Para esto pondremos nuestras credenciales en el archivo .env y las usaremos en la configuración.
- Como la configuración de la base de datos es un json, esto no nos permite usar las variables de entorno, por lo que crearemos un nuevo archivo: bd.js en la carpeta config.



Configuración

- Ahora, importaremos los valores del archivo.env
- Para esto agregamos

```
config > bd.js > <unknown>
1  require ("dotenv").config();
2
3  module.exports=
```

- Y luego copiamos el contenido de config.json y agregamos los valores del archivo .env

```
config > bd.js > <unknown> > "development"
1  require ("dotenv").config();
2
3  module.exports={
4          "development": {
5            "username": dot.env.PGUSER,
6            "password": dot.env.PGPASSWORD,
7            "database": dot.env.PGDATABASE,
8            "host": dot.env.PGHOST,
9            "dialect": dot.env.APP_BD
10         },
11         "test": {
12           "username": "root",
13           "password": null,
14           "database": "database_test",
15           "host": "127.0.0.1",
16           "dialect": "mysql"
17         },
18         "production": {
19           "username": "root",
20           "password": null,
21           "database": "database_production",
22           "host": "127.0.0.1",
23           "dialect": "mysql"
24         }
25 }
```

Configuración

- Como cambiamos el archivo de configuración, debemos modificar su llamada en el archivo index.js de la carpeta models:

```
8  const config = require(__dirname + '/../config/bd.js')[env];
9  const db = {};
```

Configuración

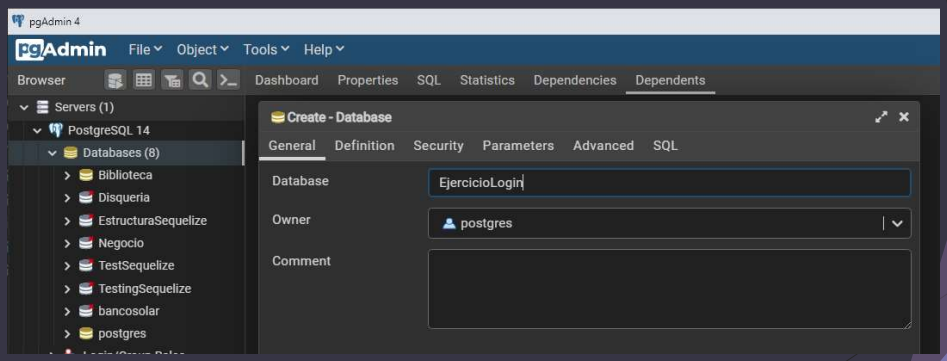
- También hay que hacer un pequeño ajuste en sequelize para que tome por defecto este archivo y no el anterior.
- Para esto, en la raíz del proyecto crearemos un nuevo archivo llamado .sequelizerc
- Aquí pondremos el siguiente código de configuración:

.sequelizerc > ...

```
1  const path = require('path');
2
3  module.exports = {
4    'config': path.resolve('config', 'bd.js')
5  };
```

BD

- Ahora. Sólo nos queda crear la base de datos “EjercicioLogin”, que es donde trabajaremos.



Modelos

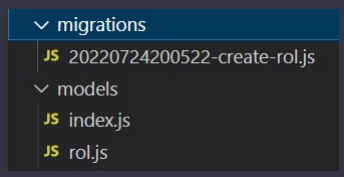
- El primer modelo que crearemos es el Rol, ya que es mas simple, esto mediante un comando de sequelize CLI.
- Recuerden que esto nos crea el modelo y la migración, la cual debemos llamar para que se concrete la creación e la tabla en la base de datos.
- Luego crearemos el modelo de Usuario, pero antes de ejecutar la migración, debemos modificar el modelo y migración para agregar:
- La PK de correo en el usuario
- La FK de Rol en el usuario
- El unique de username en usuario.

Creando Modelo Rol

- Para esto ejecutamos:

```
$ npx sequelize-cli model:generate --name Rol --attributes nombre:string
```

- Esto nos creará dos archivos:
- En la carpeta models, el archivo rol.js
- En la carpeta migrations, el archivo XXXXXXXX_create-rol.js



Modelo Rol

- Si analizamos el rol y la migración, podremos ver:
- Sólo está el campo nombre, ya que el id (PK) se crea automático en la migración.
- El archivo se llama rol.js (con minúscula) pero el modelo se llama Rol (con mayúscula).
- Aquí no cambiaremos nada.

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

Migración create-rol

- Este archivo es lo que se ejecutará en la base de datos.
- Aquí si aparecen todos los campos, incluido el id y los timestamp.
- El único cambio que haremos es el nombre de la tabla.
- Aparecer como “Rols”, lo cambiaremos a “Roles”

migrations > JS 20220724200522-create-rol.js > <unknown> > up

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

Ejecución Migración

- Ejecutaremos la migración con el comando:

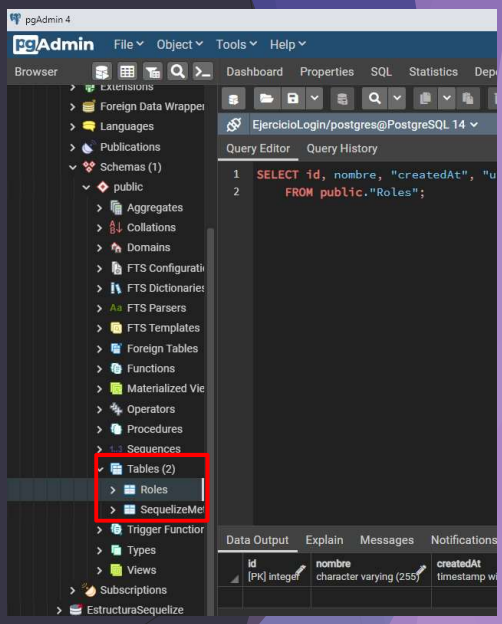
```
$ npx sequelize-cli db:migrate
```

- Esto nos creará la tabla “Roles” en la base de datos.

```
Sequelize CLI [Node: 16.15.1, CLI: 6.4.1, ORM: 6.21.3]

Loaded configuration file "config\bd.js".
Using environment "development".
== 20220724200522-create-rol: migrating =======
== 20220724200522-create-rol: migrated (0.022s)
```

- También se crea una tabla llamada SequelizeMetadata, la ignoraremos ya que es una tabla de sistema de sequelize.



Modelo Usuario

- Para crear este modelo, usaremos la siguiente línea de sequelize-cli:

```
$ npx sequelize model:generate --name Usuario --attributes email:string,username:string,nombre:string,password:string,idRol:integer
```

- Esto nos creará dos archivos: usuario.js en la carpeta models y XXXXXXX-create-usuario.js en la carpeta migrations
- Ahora, antes de ejecutar la migración, modificaremos el modelo y migración para que se ajuste a nuestro sistema.

Modelo usuario

- En este modelo tenemos que cambiar algunas cosas:definir email como PK, agregar la FK al idRol y definir username como "unique":
- Para esto cambiamos las definiciones de email, username y idRol
- Agregamos además la asociación para definir que cada usuario tiene sólo un rol.

```
16  Usuario.init({
17    email: {
18      type:DataTypes.STRING,
19      primaryKey:true
20    },
21    username: {
22      type:DataTypes.STRING,
23      unique:true
24    },
25    nombre: DataTypes.STRING,
26    password: DataTypes.STRING,
27    idRol: {
28      type:DataTypes.INTEGER,
29      references:{
30        model:"Roles", //nombre de la tabla, no del modelo
31        key:"id"
32      }
33    }
34  }, {
35    sequelize,
36    modelName: 'Usuario',
37  });
38
39
40  return Usuario;
41  };
```

Migración, create usuario

- En esta migración hacemos cambios para el mismo objetivo:
- 1.- Eliminamos el campo id y definimos email como PK
- 2.- Agregamos unique al username
- 3.- agregamos allowNull:false a todos los campos.
- 4.- agregamos la referencia (FK) al idRol
- OJO: la tabla debe llamarse Usuarios

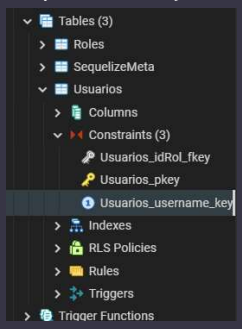
```
4   await queryInterface.createTable('Usuarios', {
5     email: {
6       allowNull: false,
7       type: Sequelize.STRING,
8       primaryKey: true,
9     },
10    username: {
11      type: Sequelize.STRING,
12      allowNull: false,
13      unique:true
14    },
15    nombre: {
16      allowNull: false,
17      type: Sequelize.STRING
18    },
19    password: {
20      allowNull: false,
21      type: Sequelize.STRING
22    },
23    idRol: {
24      type: Sequelize.INTEGER,
25      allowNull: false,
26      references:{
27        model:"Roles",//nombre de la tabla, no del modelo
28        key:"id"
29      }
30    },
```

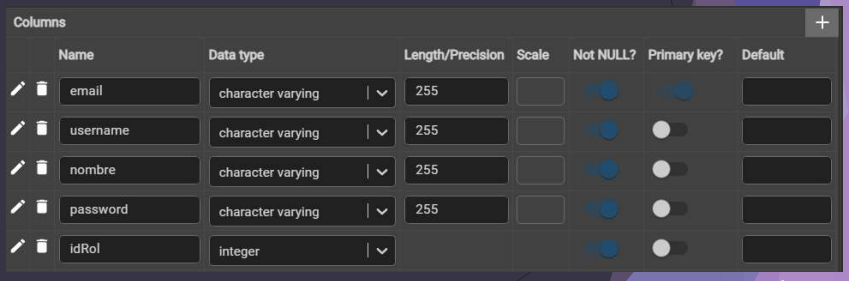
Ejecución de Migración

- Ahora ejecutamos nuevamente el comando

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

- Y la base de datos ahora debiera contener la tabla "Usuarios" con sus respectivas PK y FK.





Tenemos todo preparado

- Ahora, rellenaremos la tabla Roles, e insertaremos un usuario (para poder testear el login).
- El resto de usuarios los agregaremos con la vista/ruta de registro que crearemos mas adelante.

```
insert into "Roles" (id,nombre,"createdAt","updatedAt") VALUES (1,'admin',NOW(),NOW());
insert into "Roles" (id,nombre,"createdAt","updatedAt") VALUES (2,'usuario',NOW(),NOW());
```

- Para el usuario:

```
INSERT INTO "Usuarios"
    (email, username,nombre,password,"idRol","createdAt","updatedAt")
VALUES
    ('pachinx@gmail.com','XP','DENIS','12345',1,NOW(),NOW());
```

- Ojo que el password lo pondremos como texto, mas adelante lo encriptaremos.

Probando el Login

- El primer paso que haremos, es enviar los datos del formulario a nuestro servidor, cargaremos el usuario correspondiente, de lo contrario responderemos con un 404 (por ahora).
- Aún no tenemos registro, y además tenemos sólo un usuario, por lo que estamos limitados a eso.
- Modificaremos entonces nuestro formulario para que al presionar el botón "ingresar" nos muestre un mensaje con los datos del usuario (sin verificar el password, sólo con el objetivo de revisar si todo funciona bien).

Formulario

- Lo primero que haremos es modificar el formulario
- Es el que corresponde a la vista login.ejs (en la carpeta views)
- Agregaremos el action y el method:

```
6  <main class="form-signin w-100 m-auto formulario mt-5">
7      <form action="/login" method="post">
8          <img class="mb-4 logo" src="img/logo.png" alt="">
9          <h1 class="h3 mb-3 fw-normal">Iniciar sesión</h1>
10
```

Controlador Login

- Ahora, si lo probamos, nos dará un mensaje “OK!”, por lo tanto tenemos que cambiar el controlador para buscar el usuario y devolver sus datos:
- 1.- Dentro de la carpeta controllers, modificaremos el archivo login.js
- 2.- cargamos el modelo para usuarios:

controllers > JS login.js > ...

```
1   const {Usuario} = require("../models");
```

Controlador Login

- Ahora la función postLogin

```
7   const postLogin= async function (req,res){
8       let user;
9       try{
10          //búsqueda de usuario por su clave primaria(PK), o sea, el mail
11          user =  await    Usuario.findByPk(req.body.email);
12      }catch(err){
13          console.log("Error DB en postLogin:" + err.message);
14      }
15      //si no se encuentra nada, es porque no existe el usuario con ese mail
16      if (!user){
17          return res.send("usuario no encontrado");
18      }
19      //si verificar si coinciden los passwords, esto lo encriptaremos mas adelante.
20      if(user.password!=req.body.password){
21          return res.send("credenciales incorrectas");
22      }
23      //enviar los datos del usuario, por
24      res.send(user);
25  }
```

Configuración

- Para que funciones, nos falta agregar una configuración al sistema.
- Esta es la lectura de los datos del formulario a través del body, usando la librería body-parser
- Para esto nos vamos al archivo app.js en la raíz de nuestro proyecto, y agregamos:
- 1.- La carga de la librería o módulo:

app.js > ...

```
1   //carga de librerias--------------------------------
2   const express=require("express");
3   const router=require("./routes");
4   const bodyparser=require("body-parser");
```

Configuración

- 2.- La llamada al use en la app de express:

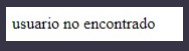
```
11  //para obetner texto del body
12  app.use(bodyparser.urlencoded({ extended: false }));
13  //importar las rutas
14  app.use("/",router);
15  module.exports=app;
```

- OJO!: SIEMPRE hacer estas llamadas o configuraciones ANTES de cargar las rutas (router)

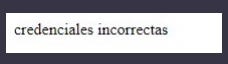
EL resultado

- Si ingresamos bien nuestro usuario y contraseña:

- De lo contrario:
- Email incorrecto:



- Password incorrecto:



Lo pendiente

- La vista de registro y toda su funcionalidad
- La vista de la página principal (index)
- La vista de la página de administración
- La encriptación del password
- La redirección a la página principal cuando iniciemos sesión
- La creación de la sesión
- Proteger las demás páginas
- Verificar el rol del usuario.
- Etc. (?)

FIN