

LABORATORIO N° 07

Desarrollo de aplicaciones Web Avanzado: Seguridad en aplicaciones con JWT.



DOCENTE:

Coello Palomino, Ricardo

CURSO:

Desarrollo de aplicaciones Web
avanzado

5 - C24 - Sección A -B-C-D

DESARROLLO DE APLICACIONES WEB AVANZADO: Seguridad en aplicaciones con JWT.

I. Capacidades

- Implementa aplicaciones usando un stack fullstack y JWT.

II. Seguridad

- En este laboratorio está prohibida la manipulación del hardware, conexiones eléctricas o de red. Así como la ingesta de alimentos y bebidas.
- Ubicar maletines y/o mochilas en lugar destinado para tal fin.
- Dejar la mesa de trabajo y la silla utilizada limpias y ordenadas.

III. Fundamento teórico

- Revise el material de la semana correspondiente antes del desarrollo del laboratorio.

IV. Normas empleadas

- *No aplica*

V. Recursos

- En este laboratorio, cada estudiante trabajará con una computadora con Windows 10.
- La instalación del software requerido se realizará en el equipo virtual.

VI. Metodología para el desarrollo de la tarea

- El desarrollo del laboratorio es individual

VII. Procedimiento

NODE.JS EXPRESS JWT AUTHENTICATION WITH MYSQL & ROLES

Desarrollaremos una aplicación Node.js Express que permita a los usuarios:

Registrarse: Crear una nueva cuenta.

Iniciar sesión: autentiqúese usando nombre de usuario y contraseña.

Acceso basado en roles: acceda a los recursos en función de roles (administrador, moderador, usuario).

TECNOLOGIA

Node.js : entorno de ejecución.

Express 4 : Marco web.

Sequelize 6 : ORM para MySQL.

MySQL 8 : Base de datos relacional.

JWT 9 : Autenticación basada en token.

bcryptjs 2 : Hashing de contraseñas.

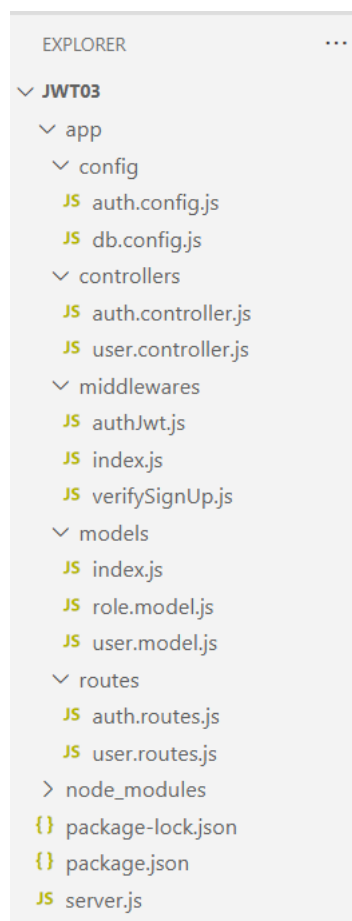
CORS 2 : Intercambio de recursos entre orígenes.

1.- Abrir el programa Visual Studio y crear la siguiente estructura de carpetas:

middlewares

role.model.js

user.routes.js



2.- Crear paquete JSON: **npm init -y**

3.- Instalar dependencias: **npm install express sequelize mysql2 cors jsonwebtoken bcryptjs**

4.- Configurar package.json:

```
"scripts": {
  "start": "node server.js"
}
```

5.- Actualización package.json de los módulos ES

```
{
  ...
  "type": "module",
  ...
}
```

CONFIGURACIÓN DE LA BASE DE DATOS

7.- Crear archivo de configuración

```
JS db.config.js X
app > config > JS db.config.js > [e] default > PORT
1 // app/config/db.config.js
2 export default {
3   HOST: "localhost",
4   USER: "root",
5   PASSWORD: "",
6   DB: "db",
7   PORT: 3306,
8   dialect: "mysql",
9   pool: {
10     max: 5,
11     min: 0,
12     acquire: 30000,
13     idle: 10000,
14   },
15 };
```

```
JS auth.config.js X
app > config > JS auth.config.js > ...
1 // app/config/auth.config.js
2 export default {
3   secret: "your-secret-key",
4 };
5
```

8.- Inicializar Sequelize y definir asociaciones de modelos:

EXPLORER

JWT03

app

config

auth.config.js

db.config.js

controllers

auth.controller.js

user.controller.js

middlewares

auth.jwt.js

index.js

verifySignUp.js

models

index.js

role.model.js

user.model.js

routes

auth.routes.js

user.routes.js

node_modules

package-lock.json

package.json

server.js

JS index.js

```

1 // Importamos Sequelize, que es el ORM que utilizaremos para interactuar
2 // con la base de datos
3 import Sequelize from "sequelize";
4
5 // Importamos la configuración de la base de datos desde un archivo
6 // externo
7 import dbConfig from "../config/db.config.js";
8
9 // Importamos los modelos de usuario y rol
10 import userModel from "./user.model.js";
11 import roleModel from "./role.model.js";
12
13 // Creamos una instancia de Sequelize con los parámetros de configuración
14 const sequelize = new Sequelize(dbConfig.DB, dbConfig.USER, dbConfig.
15   PASSWORD, {
16     host: dbConfig.HOST, // Dirección del servidor de la base
17     // de datos
18     dialect: dbConfig.dialect, // Tipo de base de datos (por
19     // ejemplo, 'mysql', 'postgres')
20     pool: dbConfig.pool, // Configuración del pool de
21     // conexiones
22     port: dbConfig.PORT, // Puerto en el que se conecta a la
23     // base de datos
24   });
25
26 // Creamos un objeto para almacenar los modelos y la instancia de
27 // Sequelize
28 const db = {};
29
30 // Asignamos Sequelize y la instancia sequelize al objeto db
31 db.sequelize = Sequelize;
32 db.sequelize = sequelize;
33
34

```

JS index.js

```

26 // Inicializamos los modelos de usuario y rol, pasándoles la instancia de
27 // Sequelize y el objeto Sequelize
28 db.user = userModel(sequelize, Sequelize);
29 db.role = roleModel(sequelize, Sequelize);
30
31 // Definimos una relación de muchos a muchos entre roles y usuarios
32 db.role.belongsToMany(db.user, {
33   through: "user_roles", // Nombre de la tabla intermedia que
34   // almacena las relaciones
35   foreignKey: "roleId", // Clave foránea en la tabla intermedia que
36   // referencia a roles
37   otherKey: "userId", // Clave foránea en la tabla intermedia que
38   // referencia a usuarios
39 });
40
41 // Definimos la relación inversa de muchos a muchos entre usuarios y roles
42 db.user.belongsToMany(db.role, {
43   through: "user_roles", // Nombre de la tabla intermedia
44   // foreignKey: "userId", // Clave foránea que referencia a usuarios
45   // otherKey: "roleId", // Clave foránea que referencia a roles
46   // as: "roles", // Alias para acceder a los roles de un
47   // usuario
48 });
49
50 // Definimos una constante con los posibles roles que se pueden asignar
51 db.ROLES = ["user", "admin", "moderator"];
52
53 // Exportamos el objeto db para que pueda ser utilizado en otras partes de
54 // la aplicación
55 export default db;
56

```

9.- Definir modelos.

EXPLORER

JWT03

app

config

auth.config.js

db.config.js

controllers

auth.controller.js

user.controller.js

middlewares

auth.jwt.js

index.js

verifySignUp.js

models

index.js

role.model.js

user.model.js

routes

JS user.model.js

```

1 // app/models/user.model.js
2 export default (sequelize, Sequelize) => {
3   const User = sequelize.define("users", {
4     username: {
5       type: Sequelize.STRING,
6       unique: true,
7     },
8     email: {
9       type: Sequelize.STRING,
10      unique: true,
11    },
12    password: {
13      type: Sequelize.STRING,
14    },
15  });
16
17   return User;
18 };

```

JS role.model.js

```

1 // app/models/role.model.js
2 export default (sequelize, Sequelize) => {
3   const Role = sequelize.define("roles", {
4     id: {
5       type: Sequelize.INTEGER,
6       primaryKey: true,
7       autoIncrement: true,
8     },
9     name: {
10      type: Sequelize.STRING,
11    },
12  });
13
14   return Role;
15 };

```

IMPLEMENTACIÓN DE FUNCIONES DE MIDDLEWARE

10.- Verificar el middleware de registro.

Comprobamos si hay nombres de usuario o correos electrónicos duplicados y valida roles.

EXPLORER

JWT03

app

config

auth.config.js

db.config.js

controllers

auth.controller.js

user.controller.js

middlewares

auth.jwt.js

index.js

verifySignUp.js

models

index.js

role.model.js

user.model.js

routes

auth.routes.js

user.routes.js

node_modules

package-lock.json

package.json

server.js

JS verifySignUp.js

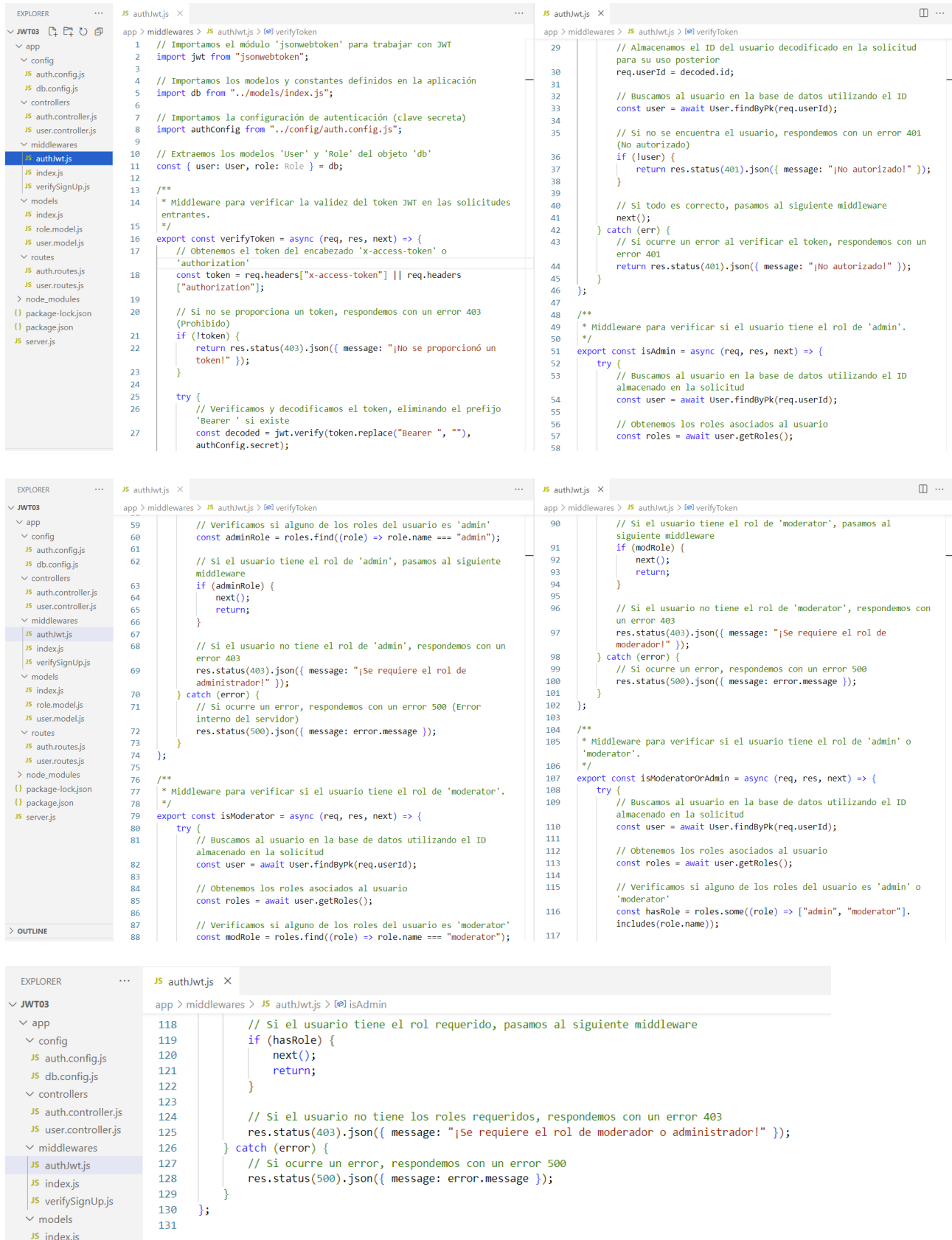
```

1 // Importamos el objeto 'db' que contiene los modelos y constantes
2 // definidos en la aplicación
3 import db from "../models/index.js";
4
5 // Extraemos la constante ROLES y el modelo User del objeto db
6 const { ROLES, user: User } = db;
7
8 /**
9  * Middleware para verificar si el nombre de usuario o el correo
10  * electrónico ya están en uso.
11  */
12 export const checkDuplicateUsernameOrEmail = async (req, res, next) => {
13   try {
14     // Buscamos un usuario con el mismo nombre de usuario
15     // proporcionado en la solicitud
16     const userByUsername = await User.findOne({
17       where: { username: req.body.username },
18     });
19
20     // Si se encuentra un usuario con ese nombre de usuario,
21     // respondemos con un error
22     if (userByUsername) {
23       return res.status(400).json({ message: "¡El nombre de usuario
24       ya está en uso!" });
25     }
26
27     // Buscamos un usuario con el mismo correo electrónico
28     // proporcionado en la solicitud
29     const userByEmail = await User.findOne({
30       where: { email: req.body.email },
31     });
32
33     // Si se encuentra un usuario con ese correo electrónico,
34     // respondemos con un error
35     if (userByEmail) {
36       return res.status(400).json({ message: "¡El correo
37       electrónico ya está en uso!" });
38     }
39
40     next();
41   } catch (error) {
42     // En caso de error en la base de datos u otro, respondemos con
43     // un error del servidor
44     res.status(500).json({ message: error.message });
45   }
46 };
47
48 /**
49  * Middleware para verificar si los roles proporcionados existen en la
50  * lista de roles permitidos.
51  */
52 export const checkRolesExisted = (req, res, next) => {
53   // Verificamos si se proporcionaron roles en la solicitud
54   if (req.body.roles) {
55     // Iteramos sobre cada rol proporcionado
56     for (const role of req.body.roles) {
57       // Si el rol no existe en la lista de roles permitidos,
58       // respondemos con un error
59       if (!ROLES.includes(role)) {
60         return res.status(400).json({ message: "¡El rol " + role +
61         " no existe!" });
62       }
63     }
64
65     // Si todos los roles son válidos o no se proporcionaron roles,
66     // pasamos al siguiente middleware
67     next();
68   }
69 };

```

11.- Middleware de autenticación JWT

Verifica tokens y verifica los roles de los usuarios.



```

// authJwt.js
import jwt from "jsonwebtoken";
import db from "../models/index.js";
import authConfig from "../config/auth.config.js";

// Extraemos los modelos 'User' y 'Role' del objeto 'db'
const { user: User, role: Role } = db;

/**
 * Middleware para verificar la validez del token JWT en las solicitudes
 * entrantes.
 */
export const verifyToken = async (req, res, next) => {
  // Obtenemos el token del encabezado 'x-access-token' o
  // 'authorization'
  const token = req.headers["x-access-token"] || req.headers
    ["authorization"];

  // Si no se proporciona un token, respondemos con un error 403
  // (Prohibido)
  if (!token) {
    return res.status(403).json({ message: "¡No se proporcionó un
      token!" });
  }

  try {
    // Verificamos y decodificamos el token, eliminando el prefijo
    // 'Bearer ' si existe
    const decoded = jwt.verify(token.replace("Bearer ", ""),
      authConfig.secret);
  }
};

// verifyToken.js
// Almacenamos el ID del usuario decodificado en la solicitud
// para su uso posterior
req.userId = decoded.id;

// Buscamos al usuario en la base de datos utilizando el ID
// (No autorizado)
const user = await User.findById(req.userId);

// Si no se encuentra el usuario, respondemos con un error 401
// (No autorizado)
if (!user) {
  return res.status(401).json({ message: "¡No autorizado!" });
}

// Si todo es correcto, pasamos al siguiente middleware
next();
} catch (err) {
  // Si ocurre un error al verificar el token, respondemos con un
  // error 401
  return res.status(401).json({ message: "¡No autorizado!" });
}
};

/**
 * Middleware para verificar si el usuario tiene el rol de 'admin'.
 */
export const isAdmin = async (req, res, next) => {
  try {
    // Buscamos al usuario en la base de datos utilizando el ID
    // almacenado en la solicitud
    const user = await User.findById(req.userId);

    // Obtenemos los roles asociados al usuario
    const roles = await user.getRoles();

    // Verificamos si alguno de los roles del usuario es 'admin'
    const adminRole = roles.find((role) => role.name === "admin");

    // Si el usuario tiene el rol de 'admin', pasamos al siguiente
    // middleware
    if (adminRole) {
      next();
      return;
    }

    // Si el usuario no tiene el rol de 'admin', respondemos con un
    // error 403
    res.status(403).json({ message: "¡Se requiere el rol de
      administrador!" });
  } catch (error) {
    // Si ocurre un error, respondemos con un error 500 (Error
    // interno del servidor)
    res.status(500).json({ message: error.message });
  }
};

/**
 * Middleware para verificar si el usuario tiene el rol de 'moderator'.
 */
export const isModeratorOrAdmin = async (req, res, next) => {
  try {
    // Buscamos al usuario en la base de datos utilizando el ID
    // almacenado en la solicitud
    const user = await User.findById(req.userId);

    // Obtenemos los roles asociados al usuario
    const roles = await user.getRoles();

    // Verificamos si alguno de los roles del usuario es 'admin' o
    // 'moderator'
    const hasRole = roles.some((role) => ["admin", "moderator"].
      includes(role.name));

    // Si el usuario tiene el rol requerido, pasamos al siguiente middleware
    if (hasRole) {
      next();
      return;
    }

    // Si el usuario no tiene los roles requeridos, respondemos con un error 403
    res.status(403).json({ message: "¡Se requiere el rol de moderador o administrador!" });
  } catch (error) {
    // Si ocurre un error, respondemos con un error 500
    res.status(500).json({ message: error.message });
  }
};

```

12.- Exportación de middleware

```

EXPLORER  JS indexjs  X
└─ JWT03
  └─ app
    └─ config
      └─ JS auth.config.js
      └─ JS db.config.js
    └─ controllers
      └─ JS auth.controller.js
      └─ JS user.controller.js
    └─ middlewares
      └─ JS authJwt.js
      └─ JS indexjs
      └─ JS verifySignUp.js

app > middlewares > JS indexjs
1  // Importa todo lo exportado desde 'authJwt.js' como un objeto llamado 'authJwt'
2  // Esto incluye funciones como verifyToken, isAdmin, isModerator, etc., si están exportadas desde ese archivo
3  import * as authJwt from "../authJwt.js";
4
5  // Importa directamente las funciones 'checkDuplicateUsernameOrEmail' y 'checkRolesExisted'
6  // desde el archivo 'verifySignUp.js'. Estas funciones probablemente validan datos del usuario durante el registro.
7  import { checkDuplicateUsernameOrEmail, checkRolesExisted } from "../verifySignUp.js";
8
9  // Reexporta los middlewares importados para que puedan ser accedidos fácilmente desde otros archivos
10 // Por ejemplo, puedes hacer: `import { authJwt, checkDuplicateUsernameOrEmail } from "../middlewares/index.js"`
11 export { authJwt, checkDuplicateUsernameOrEmail, checkRolesExisted };
12

```

CREACIÓN DE CONTROLADORES

13.- Controlador de autenticación.

Maneja el registro e inicio de sesión del usuario.

```

EXPLORER  JS auth.controller.js  ...
└─ JWT03
  └─ app
    └─ config
      └─ JS auth.config.js
      └─ JS db.config.js
    └─ controllers
      └─ JS auth.controller.js
      └─ JS user.controller.js
    └─ middlewares
      └─ JS authJwt.js
      └─ JS indexjs
      └─ JS verifySignUp.js
    └─ models
      └─ JS indexjs
      └─ JS role.model.js
      └─ JS user.model.js
    └─ routes
      └─ JS auth.routes.js
      └─ JS user.routes.js
  > node_modules
  > package-lock.json
  > package.json
  > server.js

app > controllers > JS auth.controller.js > [0] signup
1  // Importa el objeto de modelos (User, Role, etc.) desde la carpeta models
2  import db from "../models/index.js";
3
4  // Importa la librería jsonwebtoken para generar tokens JWT
5  import jwt from "jsonwebtoken";
6
7  // Importa bcryptjs para encriptar y comparar contraseñas
8  import bcrypt from "bcryptjs";
9
10 // Importa la configuración del secreto JWT desde un archivo de configuración
11 import authConfig from "../config/auth.config.js";
12
13 // Extrae los modelos User y Role desde el objeto db
14 const { user: User, role: Role } = db;
15
16 // Controlador para el registro de usuarios
17 export const signup = async (req, res) => {
18   try {
19     // Extrae los datos enviados en el cuerpo de la solicitud
20     const { username, email, password, roles } = req.body;
21
22     // Encripta la contraseña antes de guardarla en la base de datos
23     const hashedPassword = await bcrypt.hash(password, 8);
24
25     // Busca el rol "user" en la base de datos para asignarlo por defecto
26     const userRole = await Role.findOne({ where: { name: "user" } });
27
28     // Crea un nuevo usuario con los datos proporcionados y la contraseña encriptada
29     const user = await User.create({
30       username,
31       email,
32       password: hashedPassword,
33     });
34
35     // Asocia el rol encontrado al usuario (relación muchos a muchos)
36     await user.setRoles([userRole]);
37
38     // Devuelve respuesta exitosa
39     res.status(201).json({ message: "User registered successfully!" });
40   } catch (error) {
41     // Si ocurre un error, responde con código 500 y el mensaje del error
42     res.status(500).json({ message: error.message });
43   }
44 };
45
46 // Controlador para el inicio de sesión
47 export const signin = async (req, res) => {
48   try {
49     const { username, password } = req.body;
50
51     // Busca el usuario por su nombre de usuario, incluyendo sus roles
52     const user = await User.findOne({
53       where: { username },
54       include: { model: Role, as: "roles" },
55     });
56
57     // Si no se encuentra el usuario, responde con error 404
58     if (!user) {
59       return res.status(404).json({ message: "User Not found." });
60     }
61
62     // Compara la contraseña proporcionada con la almacenada (ya encriptada)
63     const passwordIsValid = await bcrypt.compare(password, user.password);
64
65     // Si la contraseña no es válida, responde con error 401
66

```

```

EXPLORER  JS auth.controller.js  ...
└─ JWT03
  └─ app
    └─ config
      └─ JS auth.config.js
      └─ JS db.config.js
    └─ controllers
      └─ JS auth.controller.js
      └─ JS user.controller.js
    └─ middlewares
      └─ JS authJwt.js
      └─ JS indexjs
      └─ JS verifySignUp.js
    └─ models
      └─ JS indexjs
      └─ JS role.model.js
      └─ JS user.model.js
    └─ routes
      └─ JS auth.routes.js
      └─ JS user.routes.js
  > node_modules
  > package-lock.json
  > package.json
  > server.js

app > controllers > JS auth.controller.js > [0] signup
65 // Si la contraseña no es válida, responde con error 401
66 if (!passwordIsValid) {
67   return res.status(401).json({
68     accessToken: null,
69     message: "Invalid Password!",
70   });
71 }
72
73 // Si la contraseña es válida, genera un token JWT que expira en 24 horas
74 const token = jwt.sign({ id: user.id }, authConfig.secret, {
75   expiresIn: 86400, // 24 horas
76 });
77
78 // Crea un array con los roles del usuario en el formato 'ROLE_ADMIN', 'ROLE_USER', etc.
79 const authorities = user.roles.map((role) => `ROLE_${role.name.toUpperCase()}`);
80
81 // Responde con la información del usuario y el token de acceso
82 res.status(200).json({
83   id: user.id,
84   username: user.username,
85   email: user.email,
86   roles: authorities,
87   accessToken: token,
88 });
89 } catch (error) {
90   // Si ocurre un error en el proceso, responde con código 500 y el mensaje del error
91   res.status(500).json({ message: error.message });
92 }
93 }
94

```


14.- Controlador del usuario

Maneja el acceso a recursos protegidos.

```

EXPLORER  ...  JS user.controller.js  X
└─ JWT03
  └─ app
    └─ config
      JS auth.config.js
      JS db.config.js
    └─ controllers
      JS auth.controller.js
      JS user.controller.js
    └─ middlewares
      JS authJwt.js
      JS index.js
      JS verifySignUp.js
    └─ models
      JS index.js
      JS role.model.js
      JS user.model.js
    └─ routes
      JS auth.routes.js

app > controllers > JS user.controller.js > ...
1  // Controlador que responde a rutas públicas (accesibles sin autenticación)
2  export const allAccess = (req, res) => {
3    |   res.status(200).send("Public Content."); // Responde con contenido público
4    | }
5  |
6  // Controlador que responde a rutas accesibles solo para usuarios autenticados
7  export const userBoard = (req, res) => {
8    |   res.status(200).send("User Content."); // Responde con contenido para usuarios comunes
9    | }
10 |
11 // Controlador que responde a rutas exclusivas para administradores
12 export const adminBoard = (req, res) => {
13 |   res.status(200).send("Admin Content."); // Responde con contenido para admins
14 | }
15 |
16 // Controlador que responde a rutas exclusivas para moderadores
17 export const moderatorBoard = (req, res) => {
18 |   res.status(200).send("Moderator Content."); // Responde con contenido para moderadores
19 | }
20 |

```

DEFINICIÓN DE RUTAS

15.- Rutas de autenticación y Rutas de usuario

```

EXPLORER  ...  JS auth.routes.js  X
└─ JWT03
  └─ app
    └─ config
      JS auth.config.js
      JS db.config.js
    └─ controllers
      JS auth.controller.js
      JS user.controller.js
    └─ middlewares
      JS authJwt.js
      JS index.js
      JS verifySignUp.js
    └─ models
      JS index.js
      JS role.model.js
      JS user.model.js
    └─ routes
      JS auth.routes.js
      JS user.routes.js
    └─ node_modules
      (l) package-lock.json
      (l) package.json
      JS server.js

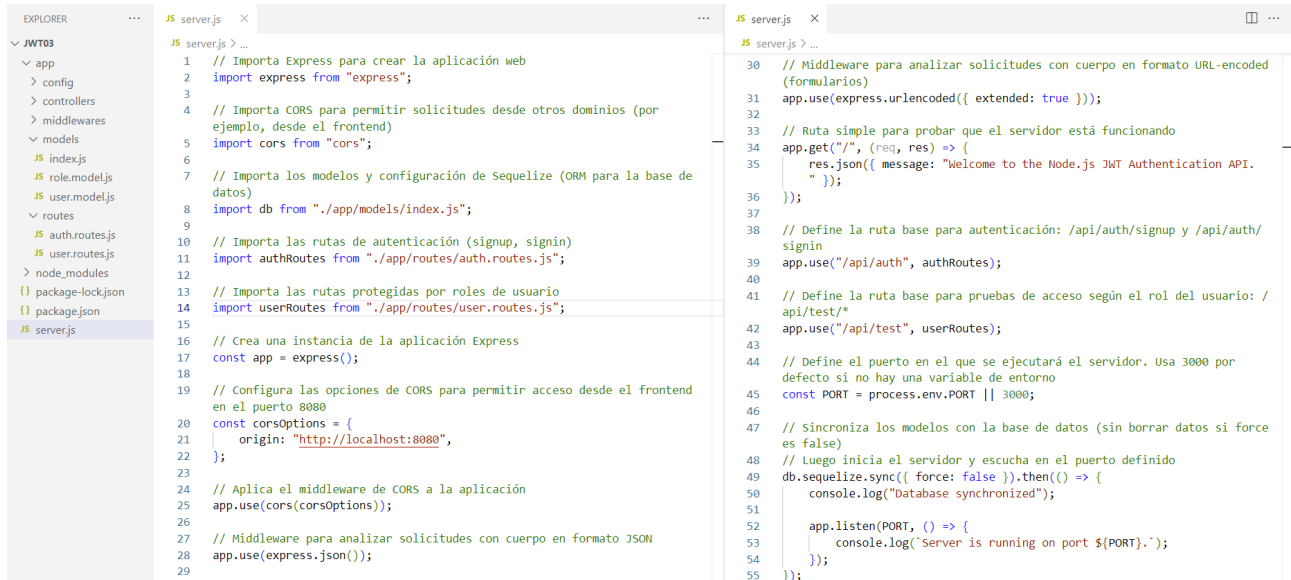
app > routes > JS auth.routes.js > ...
1  // Importa Express para definir rutas
2  import express from "express";
3
4  // Importa las funciones del controlador de autenticación
5  import { signup, signin } from "../controllers/auth.controller.js";
6
7  // Importa los middlewares que verifican datos antes de registrar
  un usuario
8  import {
9    |   checkDuplicateUsernameOrEmail, // Verifica si el username o
10   |   email ya existen
11   |   checkRolesExisted,           // Verifica si los roles
12   |   enviados son válidos
13   | } from "../middlewares/verifySignUp.js";
14
15 // Crea un router de Express para definir las rutas relacionadas
  con autenticación
16 const router = express.Router();
17
18 // Ruta para registrar un nuevo usuario (signup)
19 // Aplica dos middlewares antes de ejecutar la función signup:
20 // 1. checkDuplicateUsernameOrEmail: asegura que el username y el
21 // email no estén repetidos
22 // 2. checkRolesExisted: valida que los roles proporcionados
23 // existan en la base de datos
24 router.post("/signup", [checkDuplicateUsernameOrEmail,
25 |   checkRolesExisted], signup);
26
27 // Ruta para iniciar sesión (signin)
28 // No necesita middlewares previos, va directo al controlador
29 signin
30 router.post("/signin", signin);
31
32 // Exporta el router para poder usarlo en la configuración
  principal de rutas de la app
33 export default router;

JS user.routes.js  ●
app > routes > JS user.routes.js > ...
1  // Importa Express para crear rutas
2  import express from "express";
3
4  // Importa los controladores que manejan las respuestas según el rol del usuario
5  import {
6    |   allAccess, // Respuesta para ruta pública
7    |   userBoard, // Respuesta para usuarios autenticados
8    |   adminBoard, // Respuesta para administradores
9    |   moderatorBoard, // Respuesta para moderadores
10   | } from "../controllers/user.controller.js";
11
12 // Importa middlewares de autenticación y autorización
13 import {
14   |   verifyToken, // Verifica que el usuario esté autenticado (token
15   |   válido)
16   |   isAdmin, // Verifica que el usuario tenga rol de admin
17   |   isModerator, // Verifica que el usuario tenga rol de moderador
18   |   isModeratorOrAdmin, // Verifica que tenga uno de los dos roles
19   | } from "../middlewares/authJwt.js";
20
21 // Crea una instancia de router para definir las rutas protegidas por roles
22 const router = express.Router();
23
24 // Ruta pública: no requiere autenticación
25 router.get("/all", allAccess);
26
27 // Ruta solo para usuarios autenticados (requiere token JWT válido)
28 router.get("/user", [verifyToken], userBoard);
29
30 // Ruta solo para moderadores (requiere token + rol moderador)
31 router.get("/mod", [verifyToken, isAdmin], moderatorBoard);
32
33 // Ruta solo para administradores (requiere token + rol admin)
34 router.get("/admin", [verifyToken, isAdmin], adminBoard);
35
36 // Exporta el router para que pueda ser usado en app.js o server.js
37 export default router;

```


16.- Crear server.js

Usando la sintaxis ESMODULE , configure el servidor Express :



```

1 // Importa Express para crear la aplicación web
2 import express from "express";
3
4 // Importa CORS para permitir solicitudes desde otros dominios (por
5 // ejemplo, desde el frontend)
6 import cors from "cors";
7
8 // Importa los modelos y configuración de Sequelize (ORM para la base de
9 // datos)
10 import db from "../app/models/index.js";
11
12 // Importa las rutas de autenticación (signup, signin)
13 import authRoutes from "../app/routes/auth.routes.js";
14
15 // Importa las rutas protegidas por roles de usuario
16 import userRoutes from "../app/routes/user.routes.js";
17
18 // Crea una instancia de la aplicación Express
19 const app = express();
20
21 // Configura las opciones de CORS para permitir acceso desde el frontend
22 // en el puerto 8080
23 const corsOptions = {
24   origin: "http://localhost:8080",
25 };
26
27 // Aplica el middleware de CORS a la aplicación
28 app.use(cors(corsOptions));
29
30 // Middleware para analizar solicitudes con cuerpo en formato URL-encoded
31 // (formularios)
32 app.use(express.urlencoded({ extended: true }));
33
34 // Ruta simple para probar que el servidor está funcionando
35 app.get("/", (req, res) => {
36   res.json({ message: "Welcome to the Node.js JWT Authentication API." });
37 });
38
39 // Define la ruta base para autenticación: /api/auth/signup y /api/auth/signin
40 app.use("/api/auth", authRoutes);
41
42 // Define la ruta base para pruebas de acceso según el rol del usuario: /api/test/*
43 app.use("/api/test", userRoutes);
44
45 // Define el puerto en el que se ejecutará el servidor. Usa 3000 por
46 // defecto si no hay una variable de entorno
47 const PORT = process.env.PORT || 3000;
48
49 // Sincroniza los modelos con la base de datos (sin borrar datos si force
50 // es false)
51 db.sequelize.sync({ force: false }).then(() => {
52   console.log("Database synchronized");
53 });
54
55 app.listen(PORT, () => {
56   console.log(`Server is running on port ${PORT}.`);
57 });

```

17.- Ejecutar la aplicación: **npm start**

18.- Insertar registro en la tabla roles.

INSERT INTO roles VALUES (1, 'user', now(), now());

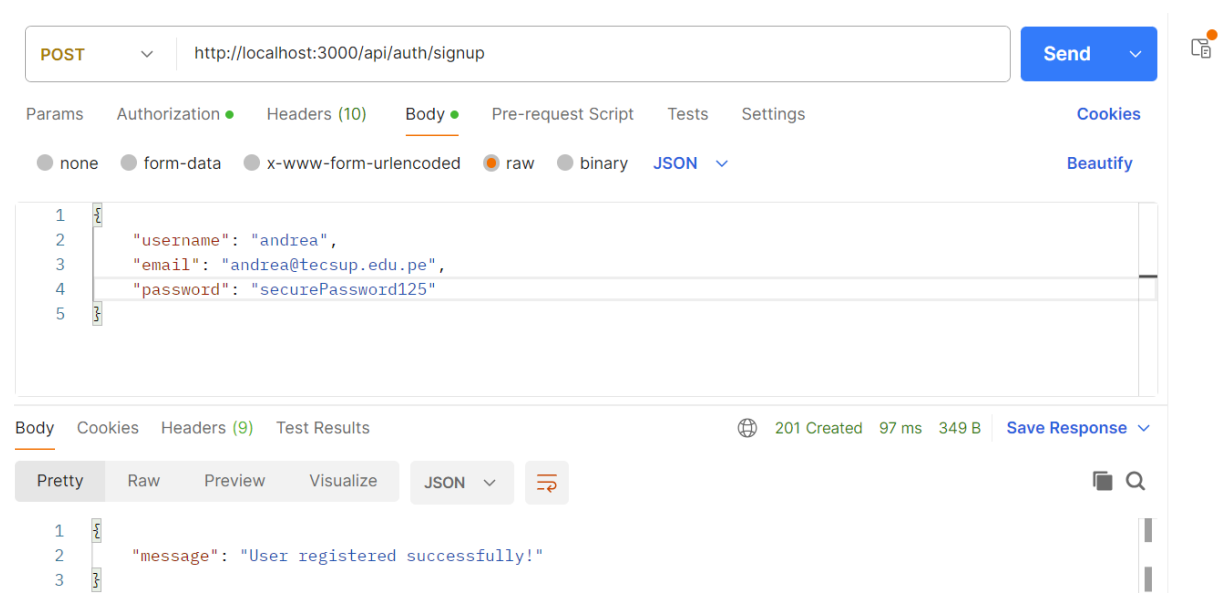
INSERT INTO roles VALUES (2, 'moderator', now(), now());

INSERT INTO roles VALUES (3, 'admin', now(), now());

PRUEBAS CON POSTMAN

19.- Ejecute Postman

20.- Registrar un usuario



POST ⌵ http://localhost:3000/api/auth/signup Send ⌵

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON ⌵ Beautify

```

1
2  "username": "andrea",
3  "email": "andrea@tecups.edu.pe",
4  "password": "securePassword125"
5

```

Body Cookies Headers (9) Test Results 201 Created 97 ms 349 B Save Response ⌵

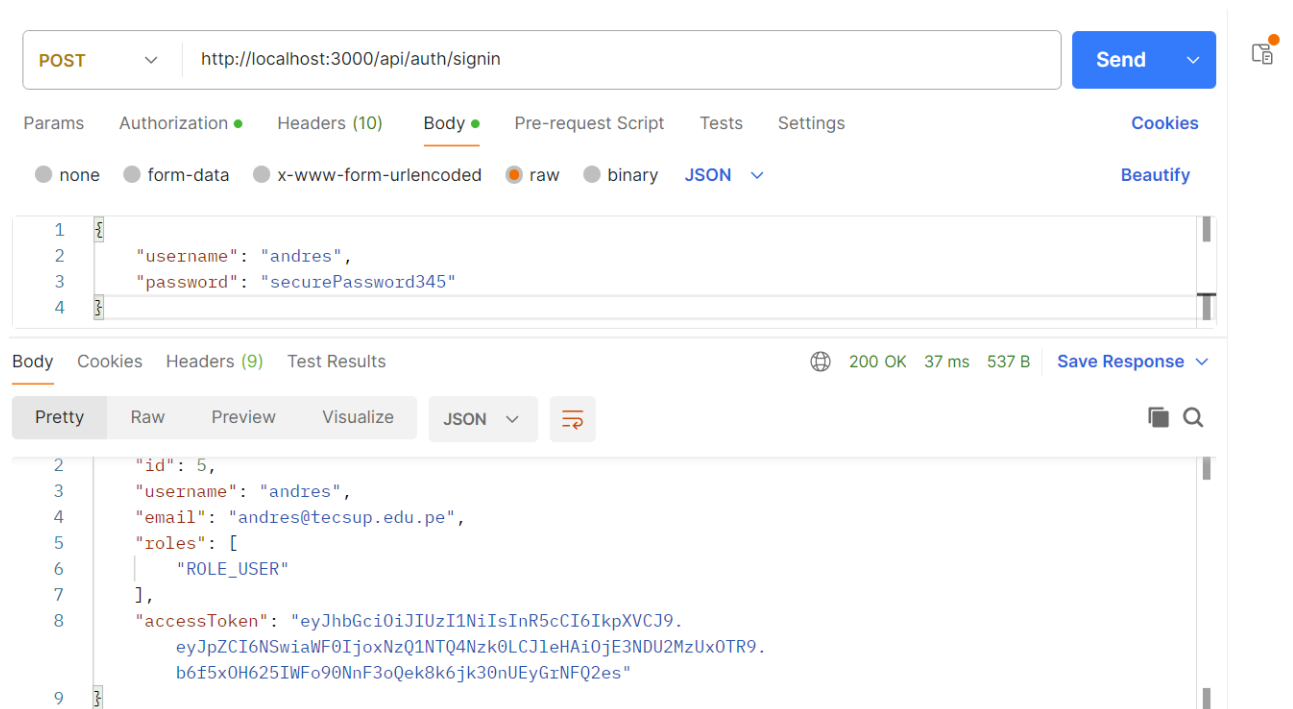
Pretty Raw Preview Visualize JSON ⌵ 🔍

```

1
2  "message": "User registered successfully!"
3

```

21.- Iniciar sesión como usuario



POST http://localhost:3000/api/auth/signin Send

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded **raw** binary JSON Beautify

```

1 {
2   "username": "andres",
3   "password": "securePassword345"
4 }

```

Body Cookies Headers (9) Test Results 200 OK 37 ms 537 B Save Response

Pretty Raw Preview Visualize JSON

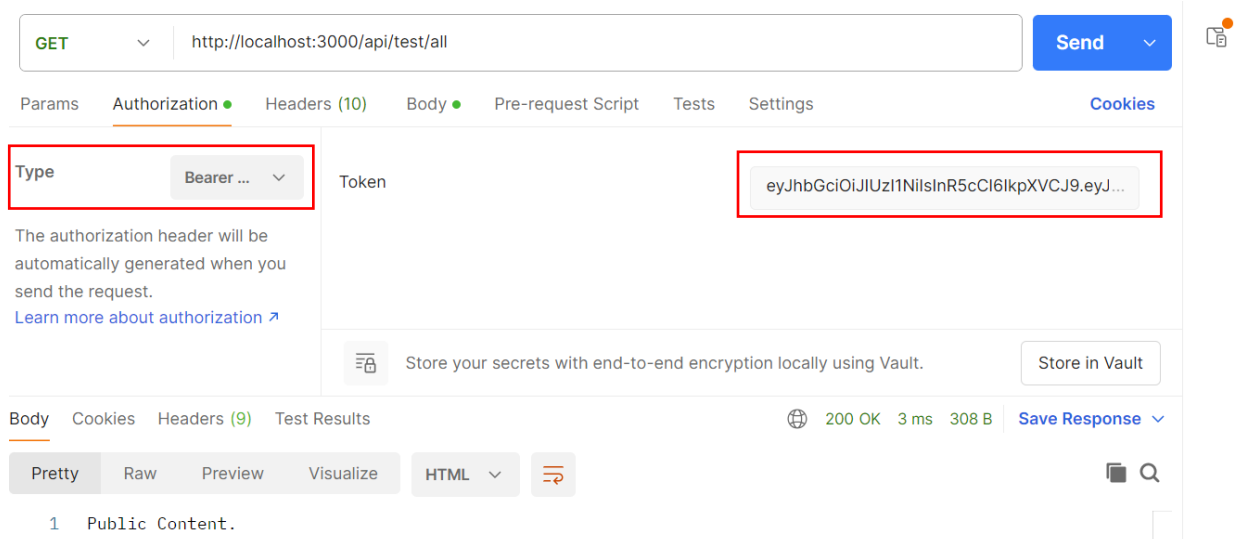
```

2 {
3   "id": 5,
4   "username": "andres",
5   "email": "andres@tecsup.edu.pe",
6   "roles": [
7     "ROLE_USER"
8   ],
9   "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NSwiaWF0IjoxNzQ1NTQ4Nzk0LCJ1eHAiOiJlE3NDU2MzUxOTR9.b6f5x0H625IWFo90NnF3oQek8k6jk30nUEyGrNFQ2es"

```

22.- Acceso a rutas protegidas.

Utilice lo recibido accessToken en el Authorization encabezado:



GET http://localhost:3000/api/test/all Send

Params **Authorization** Headers (10) Body Pre-request Script Tests Settings Cookies

Type Bearer ... Token eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NSwiaWF0IjoxNzQ1NTQ4Nzk0LCJ1eHAiOiJlE3NDU2MzUxOTR9.b6f5x0H625IWFo90NnF3oQek8k6jk30nUEyGrNFQ2es

The authorization header will be automatically generated when you send the request.
[Learn more about authorization](#)

Store your secrets with end-to-end encryption locally using Vault. Store in Vault

Body Cookies Headers (9) Test Results 200 OK 3 ms 308 B Save Response

Pretty Raw Preview Visualize HTML

```

1 Public Content.

```

Puntos finales:

- GET /api/test/all- Público
- GET /api/test/user- Usuario, Moderador, Administrador
- GET /api/test/mod- Moderador

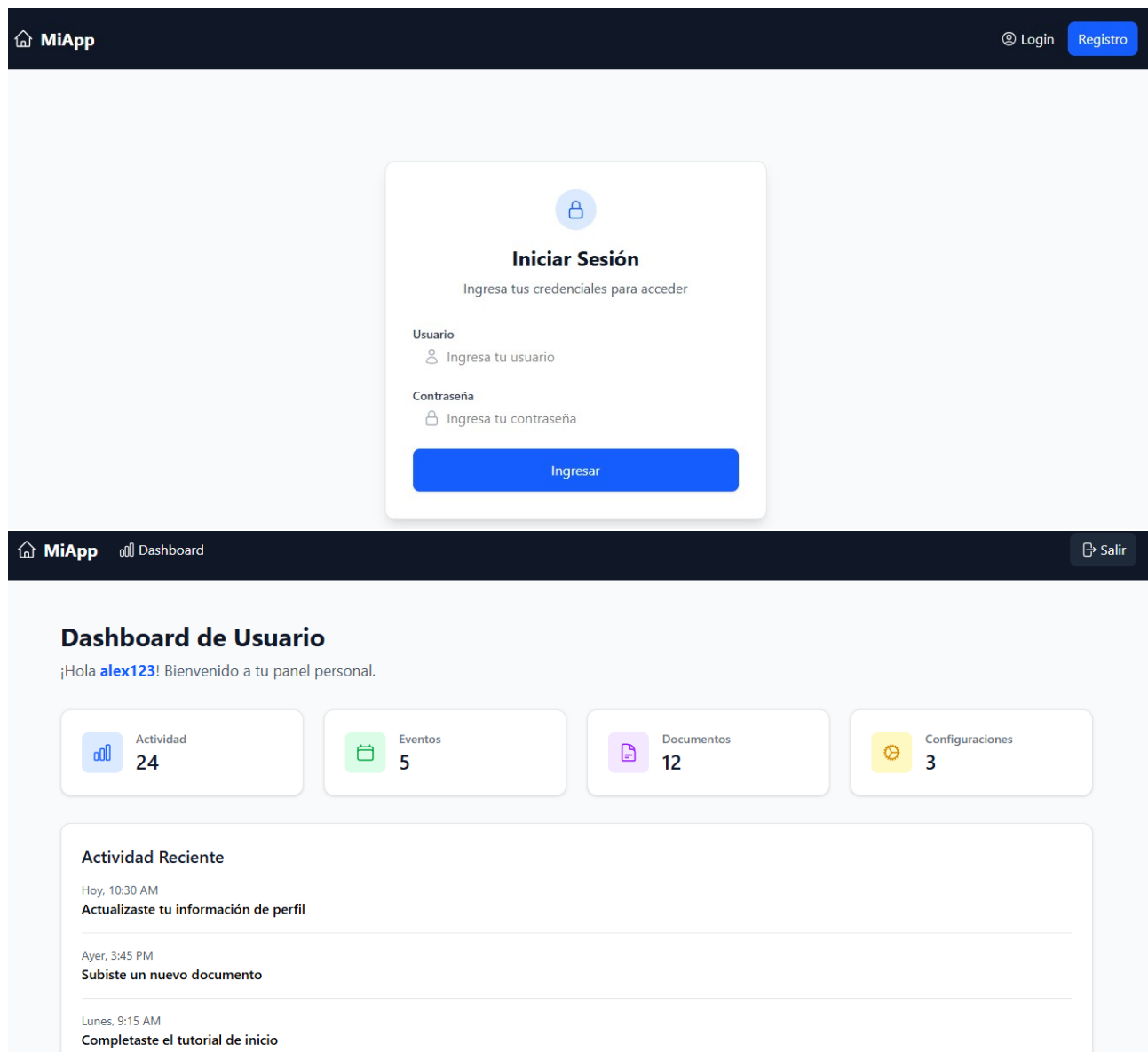
- GET /api/test/admin- Administrador

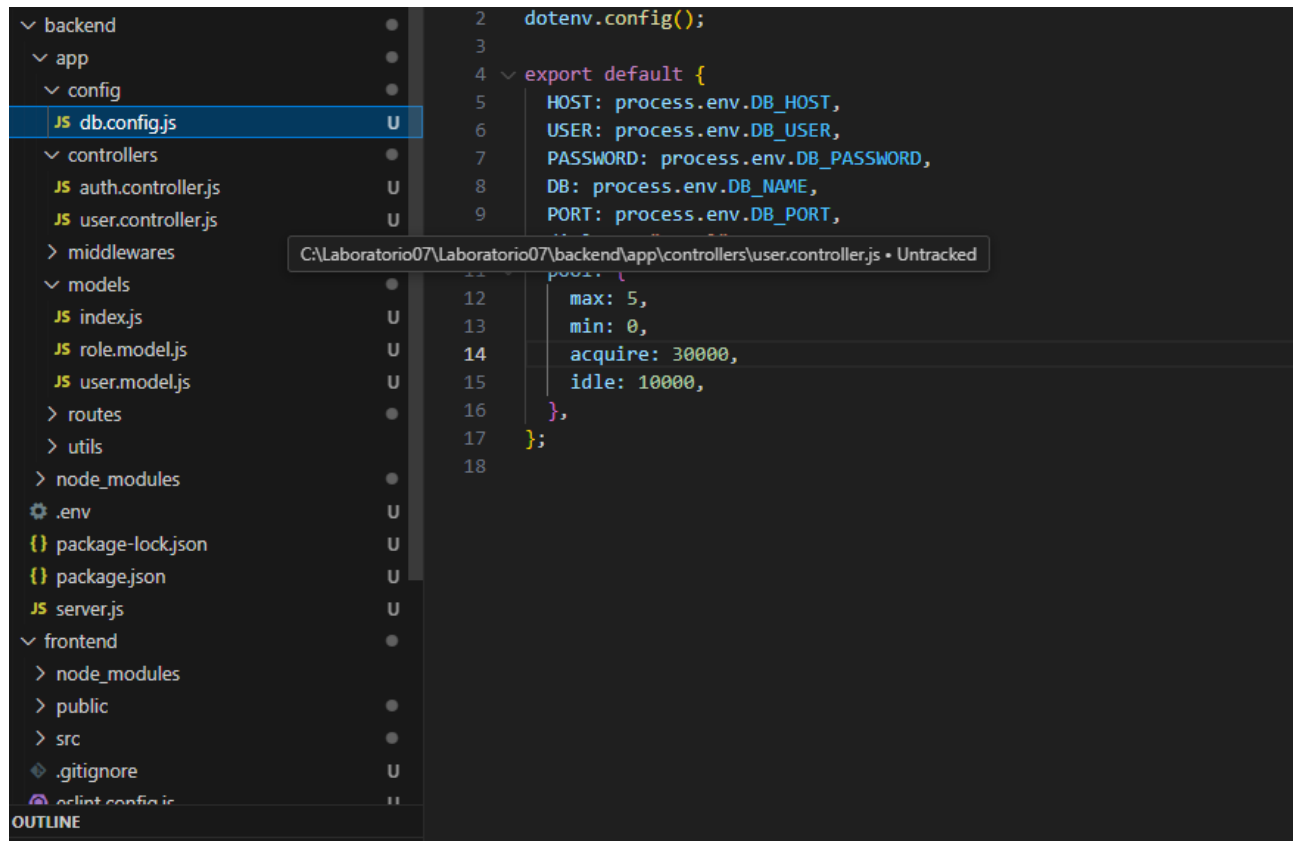
Ejercicios de aplicación

Construir una aplicación React en la que:


- Hay páginas de inicio/cierre de sesión y registro.
- Los datos del formulario serán validados por el front-end antes de enviarse al back-end.
- Dependiendo de los roles del usuario (administrador, moderador, usuario), la barra de navegación cambia sus elementos automáticamente.

Presentar capturas de pantalla de la ejecución y código de su proyecto.






```
2  dotenv.config();
3
4  export default {
5    HOST: process.env.DB_HOST,
6    USER: process.env.DB_USER,
7    PASSWORD: process.env.DB_PASSWORD,
8    DB: process.env.DB_NAME,
9    PORT: process.env.DB_PORT,
10
11    pool: {
12      max: 5,
13      min: 0,
14      acquire: 30000,
15      idle: 10000,
16    },
17  };
18
```

 Dashboard Admin Salir

Panel de Administración


Bienvenido **admin**. Gestiona toda la plataforma desde aquí.



Gestión de Usuarios

Administra todos los usuarios del sistema


[Ver detalles →](#)



Configuración del Sistema

Ajusta los parámetros globales de la plataforma

[Configurar →](#)



Estadísticas

Visualiza métricas y análisis de uso

[Ver reportes →](#)

Actividad Reciente del Sistema

FECHA	EVENTO	USUARIO
10/05/2023 14:30	Nuevo usuario registrado	usuario_nuevo
09/05/2023 09:15	Configuración actualizada	admin



Panel de Moderador

¡Hola **moderator!** Desde aquí puedes moderar comentarios, publicaciones y reportes de usuarios.

Conclusiones:

Indicar 5 conclusiones que llegó después de los temas tratados de manera práctica en este laboratorio.
