Partes de la arquitectura de la aplicación

Api express MVC

Configuración (Config)

- **mongodb.config.js** y **mysql.config.js**: Contienen la configuración para conectarse a las bases de datos **MongoDB** y **MySQL.**

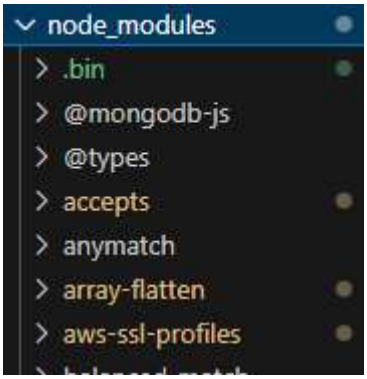
Controladores (Controllers)

- **books.js** y **booksMongo.js**: Contienen la lógica de los controladores que gestionan las peticiones HTTP relacionadas con los libros, tanto en **MySQL** como en **MongoDB.**

Base de Datos (DB)

- **books.sql**: Archivo SQL, para inicializar la base de datos en **MySQL.**
- **database.js** y **dbMongo.js**: Gestionan la conexión y las operaciones con **MySQL** y **MongoDB**, respectivamente.

Modelos (Models)

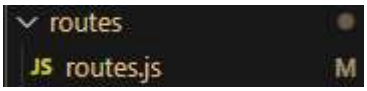


- **Library.js** y **LibraryMongo.js**: Definiciones de los modelos de datos para **MySQL** y **MongoDB.** Estos modelos se encargan de interactuar con la base de datos.

Middleware (MW)

- **auth.js**: Middleware que gestiona la autenticación y los permisos.
  - **generateToken(user)**: Crea un **JWT** (token de autenticación) con el **ID** y el **nombre de usuario.**
  - **verifyToken(req, res, next)**: Verifica si el token enviado en los **headers** es válido. Si es correcto, almacena los datos del usuario en req.user y permite continuar con la petición.

Rutas (Routes)



- **routes.js**: Define las rutas de la API y las conecta con los controladores correspondientes.

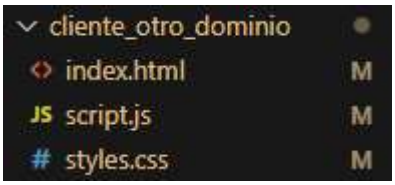
Archivo Principal

- **app.js**: Punto de entrada de la aplicación, donde se configuran los servidores, middlewares y rutas.

Dependencias

- **package.json** y **package-lock.json**: Contienen información sobre las dependencias y configuraciones del proyecto.
- **node_modules/**: Carpeta que contiene todas las dependencias instaladas con **npm.**

Cliente Web (Interfaz de Usuario)



cliente otro dominio/: Carpeta que contiene archivos relacionados con la parte visual de la aplicación:

- **index.html**: Archivo principal de la página web.
- **script.js**: Archivo JavaScript con funcionalidades para el cliente.
- **styles.css**: Hoja de estilos para la interfaz.

Adaptación del modelo de la librería de mySql a Mongo:

1) **Configuración de la Conexión:**
   a) La implementación en MySQL utiliza un pool de conexiones (mysql.createPool) para gestionar múltiples conexiones.
   b) En MongoDB, se establece una única conexión de cliente (MongoClient), y la base de datos y la colección se especifican explícitamente.

```
constructor() {
  this.client = new MongoClient(dbConfig.URL, { useNewUrlParser: true, useUnifiedTopology: true });
  this.database = "books";
  this.collection = "books";
}
```

2) **Métodos de Consulta:**
   a) Las consultas de MySQL se reemplazan con los métodos nativos de MongoDB:

- **SELECT * FROM books → collection.find({})**
- **INSERT INTO books SET ? → collection.insertOne(newBook)ç**
- **UPDATE books SET ... → collection.updateOne(query, { $set: updatedBook })**
- **DELETE FROM books ... → collection.deleteOne(query)**

listall:

```
const books = await this.collection.find({}).toArray();
```

create:

```
const result = await this.collection.insertOne(newBook);
```

update:

```
const result = await this.collection.updateOne(
  query,
  { $set: updatedBook }
);
```

delete:

```
const result = await this.collection.deleteOne(query);
```

3) **Manejo de IDs:**
    a) En MySQL, los IDs son enteros auto incrementados.
    b) En MongoDB, los IDs son ObjectId o enteros personalizados. Se añadió el método getnextID para generar IDs enteros personalizados si es necesario

```
async getnextID() {
  await this.connect();
  const lastBook = await this.collection.find().sort({ _id: -1 }).limit(1).next();
  return lastBook ? lastBook.id + 1 : 1;
}
```

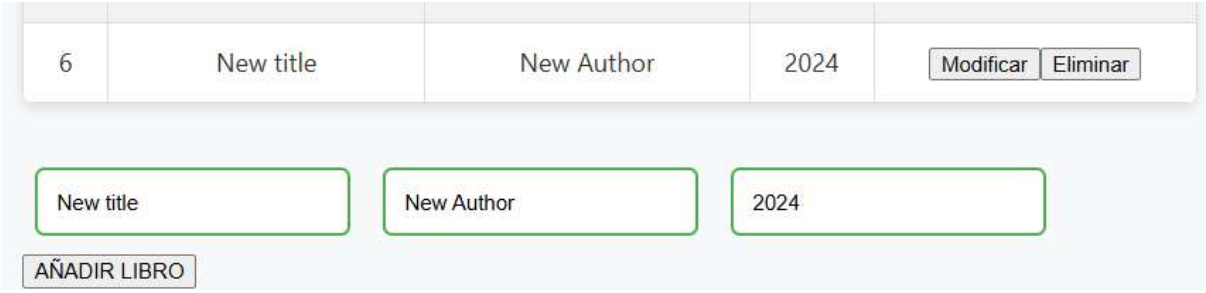
4) **Ciclo de Vida de la Conexión:**
    a) En MySQL, el pool de conexiones se gestiona globalmente y las conexiones se reutilizan.
    b) En MongoDB, la conexión se abre y cierra explícitamente para cada operación, asegurando un uso eficiente de los recursos.

```
connect = async () => {
  try {
    await this.client.connect();
    this.database = this.client.db(dbConfig.DB);
    this.collection = this.database.collection("books");
    console.log("Successfully connected to MongoDB.");
  } catch (error) {
    console.error("MongoDB connection error:", error);
    throw error;
  }
};

close = async () => {
  await this.client.close();
};
```

Captures de la funcionalidad completa utilizando Mongo (listar, adición, modificación y esborrant de libres en BBDD):

1) Listar

2) Adicción



En bbdd

4) Modificación

```
Book updated successfully
```

6	Title	Author	2024	Modificar Eliminar

```
_id: ObjectId('67baf338b7aa464a394c6803')
title : "Title"
author : "Author"
year : "2024"
id : 6
```

3) Eliminación

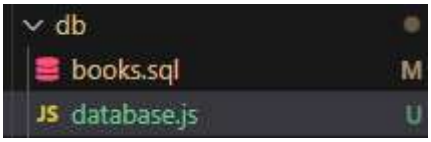
Cambios en Back end para implementar la autenticación JWT:

1) Se creo la carpeta mw y auth.js dentro:

- Dentro del auth.js:
  - **generateToken(user)** -> Genera un token JWT con información del usuario
  - **verifyToken(req, res, next)** -> Verifica que el token recibido sea válido antes de acceder a rutas protegidas.

```
const jwt = require('jsonwebtoken');
const secretKey = "mySecretKey";

// Generar un token JWT
function generateToken(user) {
    return jwt.sign({ id: user.id, username: user.username }, secretKey, { expiresIn: '1h' });
}

// Verificar un token JWT
const verifyToken = (req, res, next) => {
    const token = req.headers['authorization'];

    if (!token) {
        return res.status(403).json({ auth: false, message: 'No token provided.' });
    }

    jwt.verify(token.split(" ")[1], secretKey, (err, decoded) => {
        if (err) return res.status(401).json({ message: "Invalid Token" });
        req.user = decoded;
        next();
    });
};

module.exports = {
    generateToken,
    verifyToken
};
```

2) Creado database.js en la carpeta db para la conexión pool para auth.js



auth.js M | routes.js M | database.js U

api-express-mvc > db > database.js > ...

```
1   const mysql = require("mysql2/promise");
2   const dbConfig = require("../config/mysql.config");
3
4   // Create a connection pool
5   const pool = mysql.createPool({
6       host: dbConfig.HOST,
7       user: dbConfig.USER,
8       password: dbConfig.PASSWORD,
9       database: dbConfig.DB,
10      waitForConnections: true,
11      connectionLimit: 10,
12      queueLimit: 0,
13  });
14
15  module.exports = pool;
```

3) Modificación de las rutas/rutas.js

```
const { verifyToken, generateToken } = require('../mw/auth');
const bcrypt = require("bcryptjs");
const pool = require("../db/database");
```

```
// Protected routes
router.post('/api/books', verifyToken, books.createBook);
router.put('/api/books', verifyToken, books.updateBook);
router.delete('/api/books', verifyToken, books.deleteBook);
```

- Se importó verify Token y generates Token desde auth.js para restringir el acceso a ciertas rutas

```
// Login and registration routes
router.post("/login", async (req, res) => {
    const { username, password } = req.body;

    try {
        const [users] = await pool.query("SELECT * FROM users WHERE username = ?", [username]);
        if (users.length === 0) return res.status(401).json({ message: "Invalid Credentials" });

        const user = users[0];
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) return res.status(401).json({ message: "Invalid Credentials" });

        const token = generateToken(user);
        res.json({ token });
    } catch (err) {
        res.status(500).json({ message: "Server Error", error: err });
    }
});

router.post("/register", async (req, res) => {
    const { username, password } = req.body;

    try {
        const hashedPassword = await bcrypt.hash(password, 10);
        await pool.query("INSERT INTO users (username, password) VALUES (?, ?)", [username, hashedPassword]);
        res.status(201).json({ message: "User registered successfully" });
    } catch (err) {
        res.status(500).json({ message: "Server Error", error: err });
    }
});
```

- Implementación de login y registro
    - POST / login
        - Verifica si el usuario existe en bbdd
        - Compara la contraseña ingresada con la almacenada usando **bcrypt**.
        - Si las credenciales son correctas, genera y envía un **token JWT**.
    - POST / register
        - Hashea la contraseña con **bcrypt** antes de guardarla en la base de datos.
        - Inserta el nuevo usuario en la base de datos.

Cambios en Front end:

1) En html se añadieron formularios de login y registro:

```html
<!-- Login Form -->
<div id="login-section">
    <h2>Iniciar Sesión</h2>
    <form id="login-form">
        <input type="text" id="username" placeholder="Usuario" required>
        <input type="password" id="password" placeholder="Contraseña" required>
        <button type="button" id="loginButton">Iniciar Sesión</button>
    </form>
</div>

<!-- Registration Form -->
<div id="register-section">
    <h2>Registro de Usuario</h2>
    <form id="register-form">
        <input type="text" id="register-username" placeholder="Usuario" required>
        <input type="password" id="register-password" placeholder="Contraseña" required>
        <button type="button" id="registerButton">Registrar</button>
    </form>
</div>
```

2) Al guardar el token JWT, se guarda en el almacenamiento local y se utiliza para autorizar las solicitudes posteriores

3) En script.js:
    a) Verificación de la sesión al cargar la página y añadir event listeners

```javascript
// JWT
// Check if the user is already logged in
const token = localStorage.getItem('token');
if (token) {
    showMainContent();
    fetchBooks();
} else {
    showLoginForm();
}

// Add event listeners
document.querySelector('#loginButton').addEventListener('click', login);
document.querySelector('#logoutButton').addEventListener('click', logout);
document.querySelector('#registerButton').addEventListener('click', registerUser);
```

    b)

```
// Show the login form and hide the main content
function showLoginForm() {
    document.querySelector('#login-section').style.display = 'block';
    document.querySelector('#register-section').style.display = 'block';
    document.querySelector('#main-content').style.display = 'none';
}


// Show the main content and hide the login form
function showMainContent() {
    document.querySelector('#login-section').style.display = 'none';
    document.querySelector('#register-section').style.display = 'none';
    document.querySelector('#main-content').style.display = 'block';
}
```

c) Funciones de inicio de sesión (login)

```
// Login function
async function login() {
    const username = document.querySelector('#username').value;
    const password = document.querySelector('#password').value;

    try {
        const response = await fetch('http://localhost:5000/login', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ username, password }),
        });

        const data = await response.json();
        if (response.ok) {
            // Store the token in localStorage
            localStorage.setItem('token', data.token);
            showMainContent();
            fetchBooks();
        } else {
            alert(data.message || 'Error de autenticación');
        }
    } catch (error) {
        console.error('Error:', error);
        alert('Error de conexión');
    }
}
```

d) Función de cierre de sesión(logout)

```
// Logout function
function logout() {
    localStorage.removeItem('token');
    showLoginForm();
}
```

e) Funciones de registrar (registerUser)

```
// Registration function
async function registerUser() {
    const username = document.querySelector('#register-username').value;
    const password = document.querySelector('#register-password').value;

    try {
        const response = await fetch('http://localhost:5000/register', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ username, password }),
        });

        const data = await response.json();
        if (response.ok) {
            alert(data.message || 'Usuario registrado exitosamente');
        } else {
            alert(data.message || 'Error al registrar el usuario');
        }
    } catch (error) {
        console.error('Error:', error);
        alert('Error de conexión');
    }
}
```

f) Protección de rutas con el token JWT: En todas las rutas que requieren autenticación, como las que permiten crear, editar y eliminar libros.

```
async function deleteBook(event) {
    const token = localStorage.getItem('token');
    if (!token) {
        alert('Debes iniciar sesión para eliminar un libro.');
        showLoginForm();
        return;
    }
```

g) Función fetchBook: Ahora incluye el token en la cabecera de la petición para acceder a las rutas protegidas de la API y obtener los libros. Si el token no está presente, se redirige al usuario a la página de inicio de sesión.

```
    const response = await fetch('http://localhost:5000/api/books', {
        headers: {
            'Authorization': `Bearer ${token}`,
        },
    });
```

4) Se implementó un botón para cerrar session, eliminando el token y bloqueando el acceso a las rutas protegidas

```
<button type="button" id="logoutButton">Cerrar Sesión</button>
```

```
// Logout function
function logout() {
    localStorage.removeItem('token');
    showLoginForm();
}
```