John Carlo Garciano

DOCUMENTACIÓN

1. Arquitectura de la Aplicación (MVC)

La aplicación sigue el patrón Modelo-Vista-Controlador (MVC).

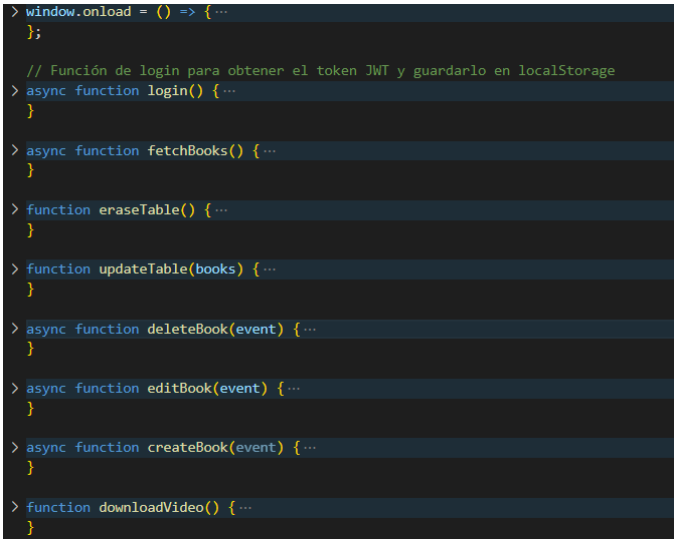
Modelo (Model)
Se encarga de interactuar con la base de datos. En el proyecto se han implementado dos modelos:

- **Library.js**: Gestiona las operaciones CRUD usando MySQL (consultas SQL, conexión mediante la librería mysql2).
  - 

- **LibraryMDB.js**: Es la adaptación del modelo para MongoDB, utilizando el cliente de Mongo y métodos como find, insertOne, updateOne y deleteOne para interactuar con la base de datos NoSQL.
  - 

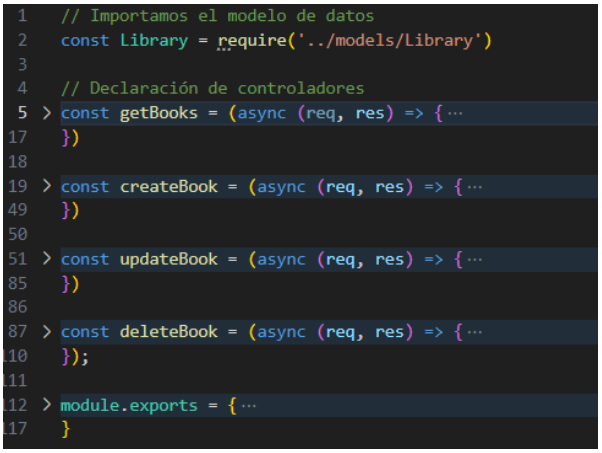
Vista (View)

La parte visual está compuesta por el archivo index.html y los scripts de JavaScript (por ejemplo, script.js) que se encargan de la interacción con el usuario. La interfaz permite visualizar los libros, así como realizar acciones de creación, edición y eliminación.



Controlador (Controller)

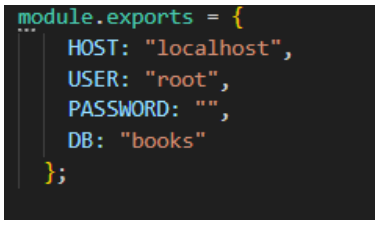
Los controladores (por ejemplo, en books.js) reciben las solicitudes HTTP, interactúan con los modelos para procesar datos y devuelven las respuestas. Las rutas definidas en routes.js direccionan las peticiones hacia los métodos correspondientes del controlador. Además, se incluyen middlewares para la autenticación con JWT.



2. Adaptación del Modelo de MySQL a MongoDB

Para aprovechar las ventajas de una base de datos NoSQL, se realizó una adaptación del modelo desarrollado originalmente para MySQL a MongoDB. Las diferencias clave son:

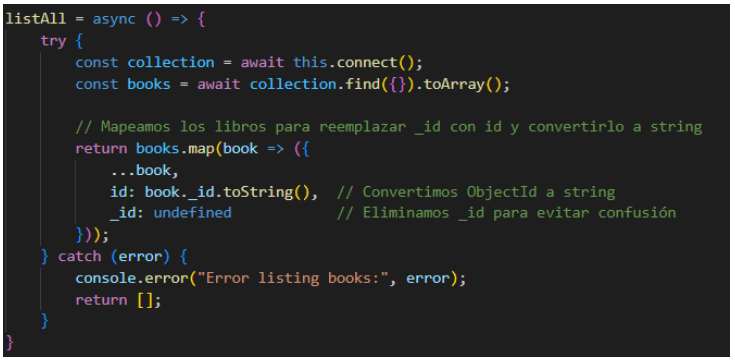
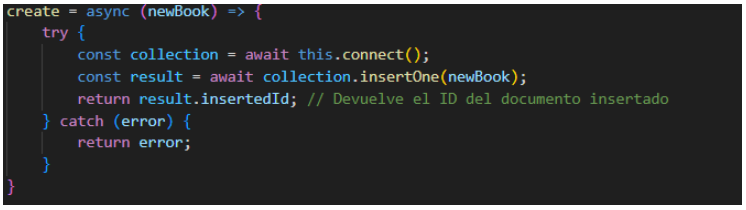
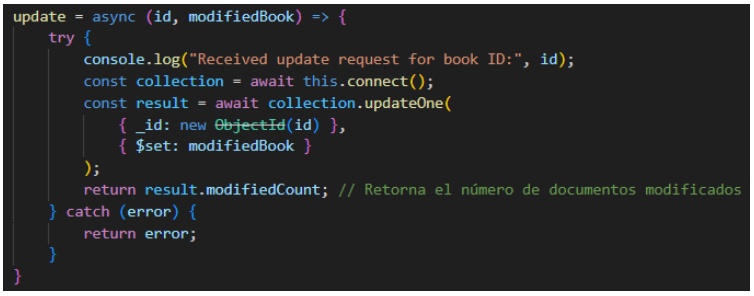
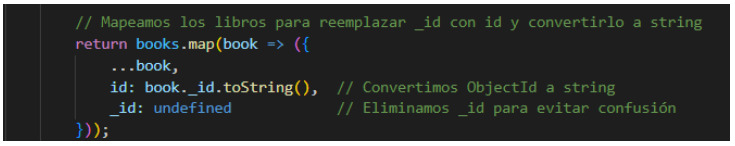
Conexión y Configuración:

- **MySQL:** Se establece una conexión usando la librería mysql2, con la configuración definida en mysql.config.js (host, usuario, contraseña y base de datos).
  - 

- **MongoDB:** Se utiliza el paquete mongodb para conectarse al clúster (según la URL y DB especificadas en mongodb.config.js).
  - 

Operaciones CRUD:

- **MySQL (Library.js):** Se ejecutan consultas SQL (ej. SELECT * FROM books, INSERT INTO books SET ?) mediante métodos promisificados.
  - 

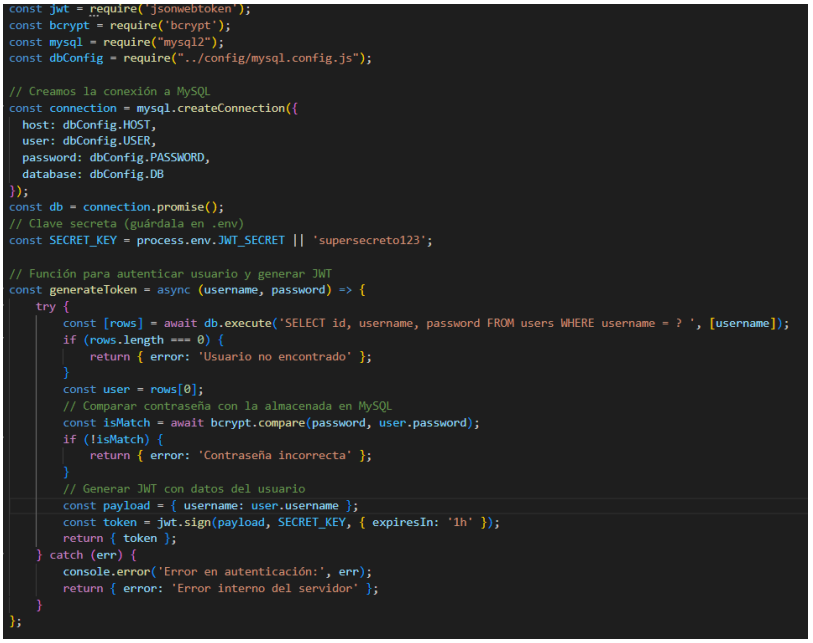
- **MongoDB (LibraryMDB.js):** Se implementan métodos asíncronos que utilizan:
  - find({}).toArray() para listar documentos.
    - 
      ```
      listAll = async () => {
          try {
              const collection = await this.connect();
              const books = await collection.find({}).toArray();

              // Mapeamos los libros para reemplazar _id con id y convertirlo a string
              return books.map(book => ({
                  ...book,
                  id: book._id.toString(),  // Convertimos ObjectId a string
                  _id: undefined            // Eliminamos _id para evitar confusión
              }));
          } catch (error) {
              console.error("Error listing books:", error);
              return [];
          }
      }
      ```
  - insertOne(newBook) para crear un nuevo registro.
    - 
      ```
      create = async (newBook) => {
          try {
              const collection = await this.connect();
              const result = await collection.insertOne(newBook);
              return result.insertedId; // Devuelve el ID del documento insertado
          } catch (error) {
              return error;
          }
      }
      ```
  - updateOne y deleteOne para actualizar y eliminar documentos, respectivamente.
    - 
      ```
      update = async (id, modifiedBook) => {
          try {
              console.log("Received update request for book ID:", id);
              const collection = await this.connect();
              const result = await collection.updateOne(
                  { _id: new ObjectId(id) },
                  { $set: modifiedBook }
              );
              return result.modifiedCount; // Retorna el número de documentos modificados
          } catch (error) {
              return error;
          }
      }
      ```
    - 
      ```
      delete = async (id) => {
          try {
              const collection = await this.connect();
              const result = await collection.deleteOne({ _id: new ObjectId(id) });
              return result.deletedCount; // Retorna el número de documentos eliminados
          } catch (error) {
              return error;
          }
      }
      ```
  - Además, se realiza una transformación en los datos de MongoDB para convertir el campo _id en id y presentarlo en formato string, evitando confusiones en el frontend.
    - 
      ```
              // Mapeamos los libros para reemplazar _id con id y convertirlo a string
              return books.map(book => ({
                  ...book,
                  id: book._id.toString(),  // Convertimos ObjectId a string
                  _id: undefined            // Eliminamos _id para evitar confusión
              }));
      ```

3. Cambios para Implementar la Autenticación JWT

La implementación de la autenticación mediante JSON Web Tokens (JWT) requirió modificaciones tanto en el backend como en el frontend:

Backend:

- Se creó el módulo auth.js donde se implementan dos funciones clave:
  - **generateToken:** Verifica las credenciales del usuario (consultando la base de datos MySQL) y, en caso de éxito, genera un token firmado con una clave secreta.



```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');
const mysql = require("mysql2");
const dbConfig = require("../config/mysql.config.js");

// Creamos la conexión a MySQL
const connection = mysql.createConnection({
  host: dbConfig.HOST,
  user: dbConfig.USER,
  password: dbConfig.PASSWORD,
  database: dbConfig.DB
});
const db = connection.promise();
// Clave secreta (guárdala en .env)
const SECRET_KEY = process.env.JWT_SECRET || 'supersecreto123';

// Función para autenticar usuario y generar JWT
const generateToken = async (username, password) => {
    try {
        const [rows] = await db.execute('SELECT id, username, password FROM users WHERE username = ? ', [username]);
        if (rows.length === 0) {
            return { error: 'Usuario no encontrado' };
        }
        const user = rows[0];
        // Comparar contraseña con la almacenada en MySQL
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) {
            return { error: 'Contraseña incorrecta' };
        }
        // Generar JWT con datos del usuario
        const payload = { username: user.username };
        const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
        return { token };
    } catch (err) {
        console.error('Error en autenticación:', err);
        return { error: 'Error interno del servidor' };
    }
};
```

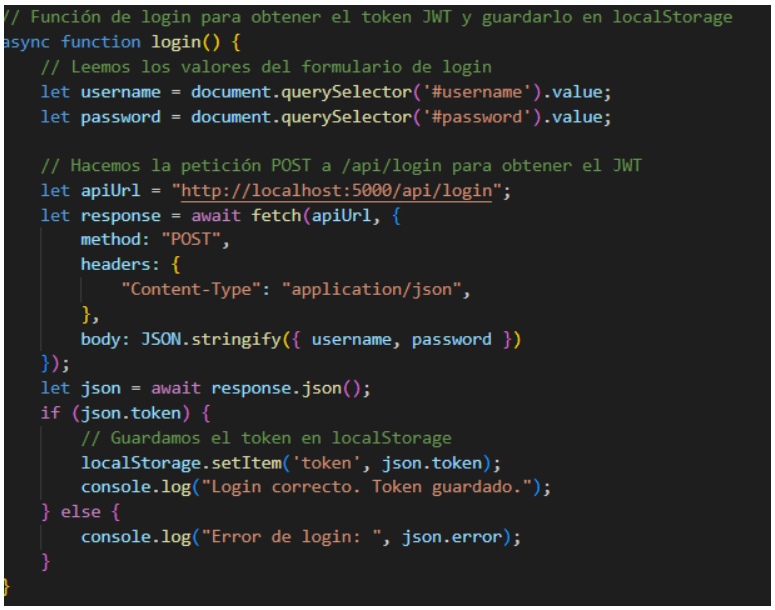
  - **jwtAuth:** Middleware que protege las rutas sensibles. Este middleware extrae el token de la cabecera Authorization, lo verifica y, de ser válido, permite el acceso a la ruta.

```
const jwtAuth = (req, res, next) => {
    const token = req.header('Authorization');
    if (!token) {
        return res.status(401).json({
            error: 'Acceso denegado. Token requerido.'
        });
    }
    try {
        const decoded = jwt.verify(token.replace('Bearer ', ''), SECRET_KEY);
        next();
    } catch (err) {
        res.status(401).json({ error: 'Token inválido o expirado.' });
    }
};

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

- En routes.js, se integran estas funciones:
  - La ruta de login **(/api/login)** utiliza **generateToken** para retornar el JWT.
  - Las rutas para crear, actualizar y borrar libros se protegen añadiendo el middleware **jwtAuth**, lo que impide el acceso a usuarios no autenticados.

```
const express = require('express');
const books = require('../controllers/books.js');
const { jwtAuth, generateToken } = require('../mw/auth.js');

const router = express.Router();

// Ruta GET pública: accesible a cualquier usuario
router.get('/api/books', books.getBooks);

// Rutas protegidas: requieren JWT en la cabecera
router.post('/api/books', jwtAuth, books.createBook);
router.put('/api/books', jwtAuth, books.updateBook);
router.delete('/api/books', jwtAuth, books.deleteBook);

// Ruta de login: recibe usuario y contraseña y retorna un JWT
router.post('/api/login', async (req, res) => {
  const { username, password } = req.body;
  const result = await generateToken(username, password);
  if (result.error) {
    return res.status(401).json(result);
  }
  res.json(result);
});

module.exports = router;
```

Frontend:

- En **script.js** se implementa la función **login**, que envía las credenciales a **/api/login** y, en caso de éxito, guarda el token en el **localStorage**.
  - 

```
// Función de login para obtener el token JWT y guardarlo en localStorage
async function login() {
    // Leemos los valores del formulario de login
    let username = document.querySelector('#username').value;
    let password = document.querySelector('#password').value;

    // Hacemos la petición POST a /api/login para obtener el JWT
    let apiUrl = "http://localhost:5000/api/login";
    let response = await fetch(apiUrl, {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ username, password })
    });
    let json = await response.json();
    if (json.token) {
        // Guardamos el token en localStorage
        localStorage.setItem('token', json.token);
        console.log("Login correcto. Token guardado.");
    } else {
        console.log("Error de login: ", json.error);
    }
}
```

- Para las operaciones protegidas (crear, modificar y eliminar libros), el token se añade en la cabecera de cada petición HTTP bajo el formato **Bearer <token>**.
- De este modo, el frontend garantiza que solo los usuarios autenticados puedan realizar modificaciones en la base de datos.