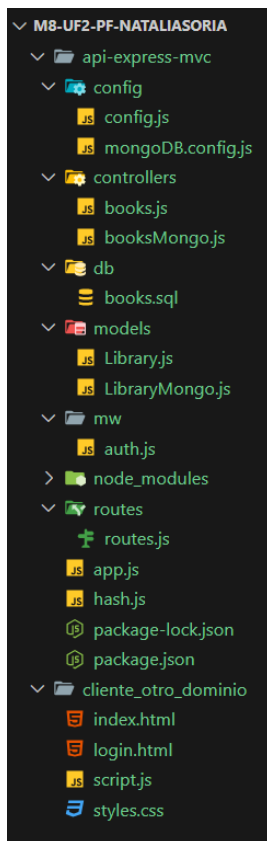


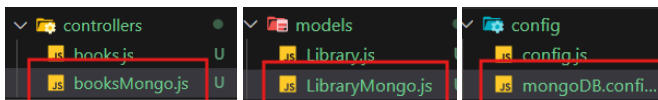
Arquitectura de la aplicacion



TAREA 1

MongoDB

Para una conexión con mongoDB utilizaremos:



En la configuración tendremos la conexión a mongoDB Compass, donde obtendremos la url de la aplicación y la base de datos que deberemos importar en el mismo mongo y que utilizaremos en el código. Para que la conexión funcione, tendrá que estar activado en todo momento en el uso de la aplicación.

```
module.exports = {
  URL: "mongodb://localhost:27017",
  DB: "books"
};
```

En la carpeta de controllers, el booksMongo se encarga de gestionar los libros, la creacion, modificacion y eliminacion de ellos. Para ello trabajaremos con varias funciones que segun el metodo que solicitemos, recogeran los datos necesarios para mostrar dicha funcionalidad.

GetBooks se encarga de obtener la lista de libros, Library es para establecer la conexión de la base de datos, y una vez conectada llamaremos a las funciones gestionadas en el mismo modelo y nos devolverá un json en base a lo pedido.

```
// Instanciamos un modelo Library
let library = new Library({});
// Lo usamos para listar libros
let books = await library.listAll();
res.json(books);
library.close();
```

Para los demás métodos realizaremos una estructura similar, adaptando a cada uno la funcionalidad que se desea aplicar:

```
// Instanciamos un modelo Library
let library = new Library({});

// Creamos un libro nuevo
const newBook = {
  title: req.body.title,
  author: req.body.author,
  year: req.body.year
};

// Usamos el modelo Library para crear libro
let created = await library.create(newBook);
```

```
let library = new Library({});

const bookID = req.body._id;
const updateBooks = {
  title: req.body.title,
  author: req.body.author,
  year: req.body.year
};
let updated = await library.update(bookID, updateBooks);
```

```
let library = new Library({});

let bookID = String(req.body._id)
let deleted = await library.delete(bookID);
```

LibraryMongo.js es el encargado de la conexión a la base de datos y de llamar o aplicar la funcionalidad correspondiente para cumplir con el método especificado.

La conexión al Mongo se hará por medio de la clase Library, en el establecemos un constructor para el uso de MongoClient y la llamada a la url y crearemos una función (connect) que nos conecta a la base de datos, en este caso “books”.

```
class Library {
  constructor() {
    this.client = new MongoClient(dbConfig.URL);
  }
  async connect() {
    try{
      await this.client.connect();
      this.database = this.client.db(dbConfig.DB);
      this.collection = this.database.collection("books");
      console.log("Successfully connected to the MongoDB database.");
    }
    catch(error){
      console.error("Error connecting to MongoDB:", error);
    }
  }
}
```

Además de la clase Library, nos encontraremos con otras funcionalidades:

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

Establece el cierre de la base de datos

```
// métodos de la clase Library
listAll = async () => {
  await this.connect();
  const books = await this.collection.find({}).toArray();
  console.log("fetched books", books);
  await this.close();
  return books;
}
```

Obtenemos todos los libros de la BD

```
create = async (newBook) => {
  try {
    await this.connect();
    const result = await this.collection.insertOne(newBook);
    return result.insertedId;
  }
  catch (error) {
    return error;
  }
};
```

Creación de un nuevo libro utilizando

como parametro "newBook" del controller "booksMongo.js", en donde nos especifica los campos que se quieren crear.

Con el metodo de update y delete haremos algo similar, con la diferencia que al querer modificar un campo o eliminar un libro en específico, deberemos utilizar como relación el ID (un campo unico que diferencia un libro de otros libros). En el mongo no utilizamos un id, si no un objectId (un conjunto de letras y numeros), por lo tanto, es un string. Para que se entienda de que estamos trabajando con un objeto, al comienzo del archivo establecemos una constante para poder utilizar, además del MongoClient, el objeto proveniente del mongoDB y de esta manera recibimos el objectId de la base de datos como un nuevo objeto:

`_id: ObjectId('67a69a20e0d566f2251df82f')` (ejemplo de objectId en mongo)

```
const { MongoClient, ObjectId } = require("mongodb");
```

```
update = async (bookID, updateBooks) => {
  await this.connect();

  const result = await this.collection.updateOne(
    { _id: new ObjectId(String(bookID)) },
    { $set: updateBooks }
  );

  await this.close();
  return result.modifiedCount;
}

delete = async (bookID) => {
  try {
    await this.connect();
    const result = await this.collection.deleteOne(
      { _id: new ObjectId(String(bookID)) }
    );
    return result.deletedCount;
  } catch (error) {
    console.error("Error delete:", error);
    return 0;
  } finally {
    await this.close();
  }
};
```

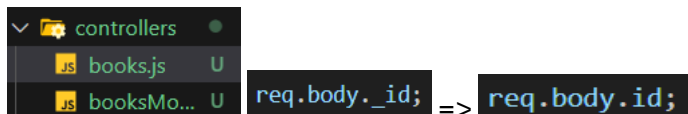
MYSQL

Con mysql las funcionalidades explicadas anteriormente son muy parecidas, lo unico que las diferencia es la conexion a la base de datos, que para este caso utilizaremos localhost como URL con su respectivo usuario, contraseña y la dataBase

```
module.exports = {  
  HOST: "localhost",  
  USER: "root",  
  PASSWORD: "",  
  DB: "books"  
};
```

(config.js)

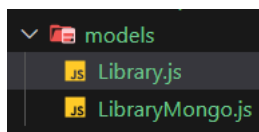
Para el controlador de mysql es el mismo procedimiento que en el de mongo, cambiando el objeto Id (_id) por id (id). este cambio lo realizamos dentro del controlador books.js (el de mysql).



```
req.body._id; => req.body.id;
```

De esta manera llamamos al id ya que mysql no tiene objeto ID.

En el archivo modelo Library.js de mysql, lo que cambia es la conexion a la base de datos y como aplicamos los metodos:



```
class Library {  
  constructor() {  
    // En el constructor, creamos una conexión a la base de datos  
    // y la guardamos en la propiedad connection de la clase  
  
    // 1.Declaramos la conexión  
    let connection = mysql.createConnection({  
      host: dbConfig.HOST,  
      user: dbConfig.USER,  
      password: dbConfig.PASSWORD,  
      database: dbConfig.DB  
    });  
  
    // 2.Abrimos la conexión  
    connection.connect(error => {  
      if (error) throw error;  
      console.log("Successfully connected to the database.");  
    });  
  }  
}
```

Trabajaremos con mysql2 para poder realizar la conexión adecuadamente

```
const mysql = require("mysql2");
```

Para los metodos realizamos consultas query para los llamados y los cambios que se quieran realizar utilizando como relacion el id en update y delete.

```

// Listar todos los libros
listAll = async () => {
  const [results, fields] = await this.connection.query("SELECT * FROM books");
  return results;
}

// Crear un nuevo libro
create = async (newBook) => {
  try {
    const [results, fields] = await this.connection.query("INSERT INTO books SET ?", newBook);
    return results.affectedRows;
  } catch (error) {
    return error;
  }
};

// Actualizar un libro
update = async (updatedBook, bookId) => {
  try {
    const [results, fields] = await this.connection.query("UPDATE books SET ? WHERE id = ?",
    [updatedBook, bookId]
    );
    return results.affectedRows;
  } catch (error) {
    return error;
  }
};

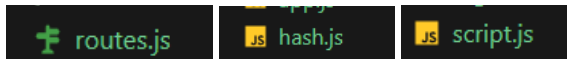
// Eliminar un libro
delete = async (bookId) => {
  try {
    const [results, fields] = await this.connection.query("DELETE FROM books WHERE id = ?",
    [bookId]
    );
    return results.affectedRows;
  }
};

```

TAREA 2

Una vez, habiendo entendido las conexiones a la base de datos de mysql y mongo, realizaremos una autentificacion por usuario, contraseña y creacion de token, utilizando como base una conexion mysql.

Archivos modificados para esta tarea:



También crearemos un nuevo html para el login de usuario:



Iniciar sesión

Username

Password

INICIAR SESIÓN

ACCEDER SIN USUARIO

El objetivo de esta tarea es que la persona pueda acceder con o sin usuario a la biblioteca. Si decide ingresar con usuario y contraseña, se mostrará la biblioteca de libros y podrá editar, crear o borrar. En caso de querer acceder sin usuario se verá la biblioteca de libros, pero "no" podrá editar, crear o borrar.

Hash.js

El archivo **hash.js** es el encargado de convertir una contraseña en un hash utilizando “bcrypt”. Es una forma segura de almacenar contraseñas en las bases de datos en lugar de guardar una contraseña en texto plano (“1234”) se guardan en formato hash y se entiende que la que contraseña que pongamos (1234) sera la de ese formato.

```
const bcrypt = require('bcrypt');

async function generarHash() {
  const password = "1234";
  const saltRounds = 10;

  const hashedPassword = await bcrypt.hash(password, saltRounds);
  console.log("Contraseña hasheada:", hashedPassword);
}

generarHash();
```

Routes.js

En rutas se manejan la mayoría de conexiones inportando dichos modulos.

En este caso agregaremos, ademas de los ya existentes, los modulos necesarios para la autenticacion del usuario

```
const express = require('express');
const books = require('../controllers/books.js');

const Library = require('../models/Library');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

const router = express.Router();
const library = new Library();
```

Importaremos Library para la conexión con la base de datos y crearemos una instancia del mismo.

Bcrypt para comparar y generar contraseñas cifradas.

Y jwt para generar el token.

Crearemos una ruta para el login (/api/login)

```
router.post('/api/login', async (req, res) => {
  const { username, password } = req.body;
```

Trabajaremos con una funcion asincrona para las llamadas a la base de datoS, y almacenaremos en una constante el usuario y contraseña recibidos desde la BD

En un bloque try-catch analizaremos las siguientes secciones:

```
try {
  // Verificar si el usuario existe
  const [users] = await library.connection.query('SELECT * FROM users WHERE username = ?', [username]);
  if (users.length === 0) {
    return res.status(401).json({ message: 'Usuario no encontrado' });
  }

  const user = users[0];

  // Comparar la contraseña con bcrypt
  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) {
    return res.status(401).json({ message: 'Contraseña incorrecta' });
  }

  // Si todo es correcto, generar el token

  const SECRET_KEY = process.env.JWT_SECRET || 'supersecreto123';

  const payload = { username: user.username };
  const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });

  res.json({ token });
} catch (err) {
```

Mediante una consulta sql verificaremos que el usuario aplicado sea igual al de la base de datos => `await library.connection.query('SELECT * FROM users WHERE username = ?', [username]);`

Si no hay relacion, mostrara el error 401, en caso de encontrar un usuario lo

guardaremos en una lista y obtendremos el primero => `const [users]` => `const user = users[0];`

Compararemos, utilizando bcrypt, la contraseña almacenada en la base de datos, al igual que el usuario, y la contraseña ingresada => `bcrypt.compare(password, user.password);`

En caso de no ser correcta, mostrara un error 401, junto con un mensaje

Si todo es correcto, tanto el usuario como contraseña, el siguiente paso es generar el token.

Primero deberemos crear una constante que será la clave usada para firmar los tokens. Primero se fija si existe una clave en las variables de entorno, y si no utiliza la asignada por defecto => `const SECRET_KEY = process.env.JWT_SECRET || 'supersecreto123';`

Se define payload como objeto del usuario ingresado correspondiente al de la base de datos => `const payload = { username: user.username };`

Creamos un token asignando una firma (jwt.sign) al usuario, junto con clave secreta y le aplicamos un límite de 1h. Al finalizar le mostramos al cliente el token generado.

```
const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });  
res.json({ token });
```

Script.js

Le hemos añadido tres funciones al script para que adapte los cambios adecuados al login:

Empezaremos con loginUser()

```
async function loginUser(event) {  
  event.preventDefault();  
  
  const username = document.querySelector("#username").value;  
  const password = document.querySelector("#password").value;  
  
  let apiUrl = "http://localhost:5000/api/login";  
  let userData = { username, password };  
  
  try {  
    let response = await fetch(apiUrl, {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify(userData),  
    });  
  
    let data = await response.json();  
  
    if (response.ok) {  
      console.log("Token recibido:", data.token);  
      localStorage.setItem("token", data.token);  
      window.location.href = "index.html";  
    } else {  
      // ...  
    }  
  }  
}
```

Obtenemos los valores de username y password ingresados por el usuario:

```
("#username").value;  
("#password").value;
```

La solicitud del servidor la hacemos en base a la url en la que manejamos la parte de autenticación trabajada en routes.js anteriormente => `let apiUrl = "http://localhost:5000/api/login";`

(Esta api corresponde a donde se enviará la solicitud de login)

Creamos un objeto con los datos del usuario obtenidos => `let userData = { username, password };`

Dentro del bloque try-catch, enviaremos la solicitud al servidor, utilizando la petición POST a la api:

```
let response = await fetch(apiUrl, {  
  method: "POST",  
  headers: { "Content-Type": "application/json" },  
  body: JSON.stringify(userData),  
});
```

Transformamos el objeto en un json => `let data = await response.json();`

Y verificamos que el login haya sido exitoso, si salió todo bien, mostrara el token recibido por consola, guardaremos el token en el localStorage y por último rediregiremos al usuario a la página de la biblioteca:


```
if (response.ok) {
  console.log("Token recibido:", data.token);
  localStorage.setItem("token", data.token);
  window.location.href = "index.html";
}
```

La funcion logoutUser() es la encargada de borrar el token cuando se cierra la sesion.

```
function logoutUser() {
  // Eliminar el token del localStorage
  localStorage.removeItem("token");
  window.location.href = "login.html";
}
```

Eliminamos el token guardado en el localStorage cuando el usuario haga clic en el boton de cerrar sesion y se redirige al login nuevamente. De esta manera el token no quedara siempre guardado. Lo que sucedía antes era que al ingresar con usuario y volver atrás, el token quedaba guardado y al darle clic al boton de acceder sin usuario accedia igualmente con un token entonces la funcion de checkUser no cumplia el proposito de ingresar sin usuario (no mostrar botones de modificar y eliminar).

1_

A login form with a text input containing 'admin', a password input with masked characters '....', and two green buttons: 'INICIAR SESIÓN' and 'ACCEDER SIN USUARIO'. A red arrow points to the 'INICIAR SESIÓN' button.

2_

A user profile card with the number '1851' and two buttons: 'Modificar' and 'Eliminar'. A red box highlights these buttons. To the right, a console log shows a message: 'Token enviado en script.js:63' and a long alphanumeric string.

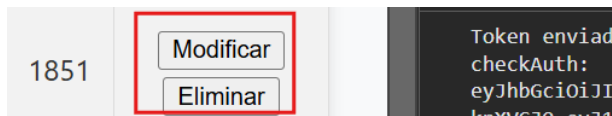
3_



4_

The login form from step 1, but now the 'ACCEDER SIN USUARIO' button is highlighted with a red arrow. To the right, a console log shows a message: 'Token enviado en script.js:63' and a long alphanumeric string.

5_



De esta manera, con esta nueva función se reinicia cada vez al darle a cerrar sesión.

La última función agregada al script es la de `checkAuth()`, esta función es la encargada de verificar si existe un token.

```
async function checkAuth() {  
  const token = localStorage.getItem('token') || sessionStorage.getItem('token');  
  const actionsSection = document.getElementById("actions-section");  
  console.log("Token enviado en checkAuth:", token);  
  
  if (!token) {  
    if (actionsSection) actionsSection.style.display = "none";  
    return;  
  }  
  
  // Verificar si el token es valido  
  try {  
    let response = await fetch("http://localhost:5000/api/books", {  
      method: "GET",  
      headers: {  
        "Authorization": `Bearer ${token}`  
      }  
    });  
  
    if (!response.ok) {  
      console.log("Token inválido. Cerrando sesión...");  
      localStorage.removeItem("token");  
  
      if (actionsSection) actionsSection.style.display = "none";  
    }  
    else {  
      if (actionsSection) actionsSection.style.display = "block";  
    }  
  }  
}
```

Obtenemos el token almacenado en el `localStorage` => `localStorage.getItem('token')`

Seleccionamos la sección del formulario en el que el usuario puede crear un nuevo libro si existe un token. Por defecto estará oculto hasta que se confirme que haya un token:

```
<div id="actions-section" style="display: none;">  
  <form id="book-form">  
    <input type="hidden" id="book-id">  
    <input type="text" id="book-title" placeholder="Título" required>  
    <input type="text" id="book-author" placeholder="Autor" required>  
    <input type="number" id="book-year" placeholder="Año" required>  
    <button type="button" id="createButton">AÑADIR LIBRO</button>  
  </form>  
</div>
```

Mostramos el token recibido, en caso de no haber es `null` => `console.log("Token enviado en checkAuth:", token);`

Si no hay token no se muestra la sección de crear:

```
if (!token) {  
  if (actionsSection) actionsSection.style.display = "none";  
  return;  
}
```

Dentro del `try` realizaremos la verificación del token:

```
y {
let response = await fetch("http://localhost:5000/api/books", {
  method: "GET",
  headers: {
    "Authorization": `Bearer ${token}`
  }
})
```

El token es enviado dentro del header

Si el token es invalido, muestra un mensaje, se borra el token incorrecto y el formulario permanece oculto

```
if (!response.ok) {
  console.log("Token inválido. Cerrando sesión...");
  localStorage.removeItem("token");

  if (actionsSection) actionsSection.style.display = "none";
}
```

En caso de que el token sea válido mostrara la seccion de crear libros

```
else {
  if (actionsSection) actionsSection.style.display = "block";
}
```

Las demas funcionalidades (update y delete) se han modificado dentro de las funciones ya existentes de update y delete, hemos agregado la condicion correspondiente a la validacion del token, de esta manera sucedera lo mismo que la funcionalidad de crear (si hay token se muestra y si no, no se muestra):

```
if (localStorage.getItem('token')) {
  let buttonEdit = document.createElement('button');
  buttonEdit.innerHTML = "Modificar";
  buttonEdit.addEventListener('click', editBook);
  celdaAcciones.append(buttonEdit);

  let buttonDelete = document.createElement('button');
  buttonDelete.innerHTML = "Eliminar";
  buttonDelete.addEventListener('click', deleteBook);
  celdaAcciones.append(buttonDelete);
}
```