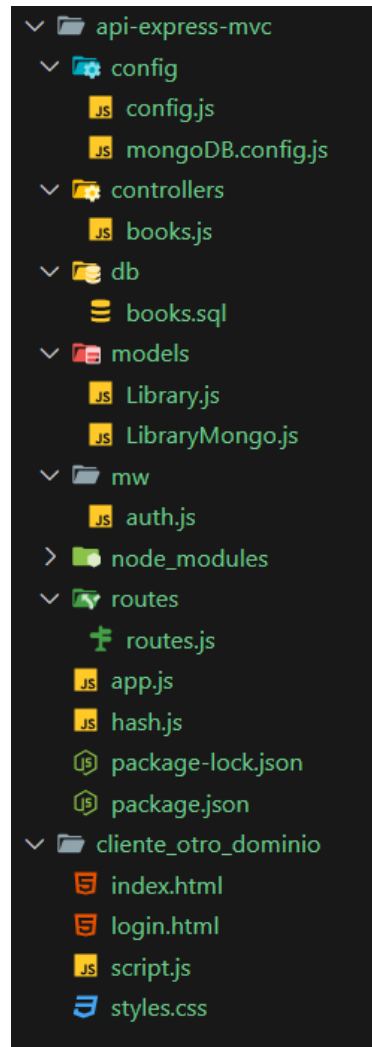


## Arquitectura de la aplicacion

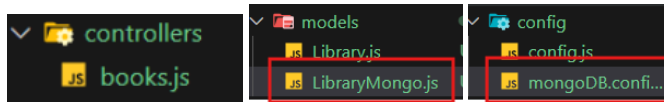


## TAREA 1

### MongoDB

(El books.js sirve para ambas conexiones, trabajaremos de manera agnóstica y de esta manera se puede adaptar el mismo código para las dos bases de datos)

Para una conexión con mongoDB utilizaremos:



En el **config**. 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 books se encarga de gestionar los libros, la creación, modificación y eliminación de ellos. Para ello trabajaremos con varias funciones que según el método que solicitemos, recogerán los datos necesarios para mostrar dicha funcionalidad.

**Library** es para establecer la conexión de la base de datos, y una vez conectada llamaremos a las funciones gestionadas en el mismo **modelo**.

```
// 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 realizar consultas a la misma, para cumplir con el método requerido.

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 parámetro "newBook" del controller “books.js”, en donde nos especifica los campos que se quieren crear.

Con el método 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 único que diferencia un libro de otros libros). En el mongo no

utilizamos un id, si no un objectId (un conjunto de letras y números), 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 único que las diferencia es la conexión a la base de datos, que para este caso utilizaremos localhost como URL con su respectivo usuario, contraseña y dataBase

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

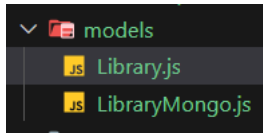
(config.js)

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 .

```
▼ controllers
  JS books.js 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 conexión a la base de datos y como aplicamos los métodos:



```
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 métodos realizamos consultas query para los llamados y los cambios que se quieran realizar utilizando como relación 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;
  }
}
```

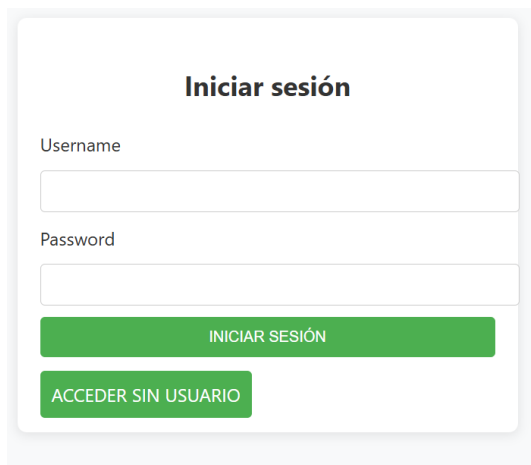
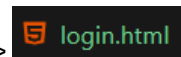
## TAREA 2

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

Archivos modificados para esta tarea:



También crearemos un nuevo html para el login de usuario =>

A login form with a light gray border. At the top, it says 'Iniciar sesión'. Below that are two input fields: 'Username' and 'Password'. Under the 'Password' field is a green button that says 'INICIAR SESIÓN'. At the bottom of the form is another green button that says '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) será 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();
```

## Auth.js

Manejaremos dos funciones principales

1\_ `const generateToken`

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

// Función para generar el token
const generateToken = (user) => {
  const payload = { username: user.username };
  return jwt.sign(payload, SECRET_KEY, { expiresIn: '24h' });
};
```

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';`

Recibiendo como objeto el “user”.

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 24h.

2\_ `const jwtAuth`

```
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);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ error: 'Token inválido o expirado.' });
  }
};
```

En esta funcion se verifica si la solicitud recibe un token en el encabezado:

```
const jwtAuth = (req, res, next) => {  
  const token = req.header('Authorization');
```

Verificamos el token, lo decodificamos y lo guardamos en req.user y continuamos con la ejecución

```
const decoded = jwt.verify(token.replace('Bearer ', ''), SECRET_KEY);  
req.user = decoded; } => next();
```

## **Routes.js**

En rutas se manejan la mayoría de conexiones importando dichos módulos.

En este caso agregaremos, además de los ya existentes, los módulos necesarios para la autenticación del usuario

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

JwtAuth y generateToken para validar y 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 función asíncrona 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 token = generateToken(user);

  res.json({ token });
} catch (err) {
  console.error('Error en el login:', err);
  res.status(500).json({ message: 'Error interno en el servidor' });
}

```

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 relación, 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 y la contraseña ingresada => `bcrypt.compare(password, user.password);`

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

Por último, si la validación del usuario y contraseña es correcta, procedemos a llamar a la función de generateToken para crear el token y mostrarlo en formato json al cliente:

```

// Si todo es correcto, generar el token
const token = generateToken(user);

res.json({ token });

```

Al finalizar validaremos si el token existe en las rutas de los métodos correspondiente, esto también lo validaremos desde los métodos del script:

```

router.get('/api/books', books.getBooks);
router.post('/api/books', jwtAuth, books.createBook);
router.put('/api/books', jwtAuth, books.updateBook);
router.delete('/api/books', jwtAuth, books.deleteBook);

```

## Script.js

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

## 1\_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 autenticacion trabajada en routes.js anteriormente => `let apiUrl = "http://localhost:5000/api/login";`

(Este 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 al 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 redirigiremos al usuario a la página de la biblioteca:

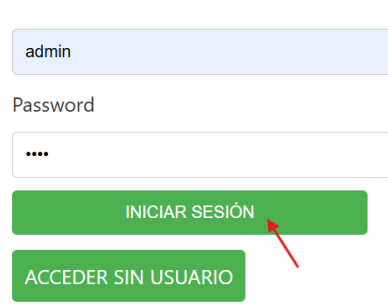
```
if (response.ok) {
  console.log("Token recibido:", data.token);
  localStorage.setItem("token", data.token);
  window.location.href = "index.html";
}
```

2\_La función 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 botón de cerrar sesión 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 botón de acceder sin usuario accedía igualmente con un token entonces los métodos PUT, POST y DELETE se encontraban disponibles porque el token seguía activo. Por lo tanto, no cumplía con el requisito de ingresar sin usuario (no poder modificar los libros)

1\_



admin

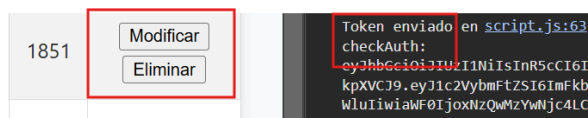
Password

....

INICIAR SESIÓN

ACCEDER SIN USUARIO

2\_



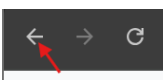
1851	Modificar	Eliminar
------	-----------	----------

Token enviado en script.js:63

checkAuth:

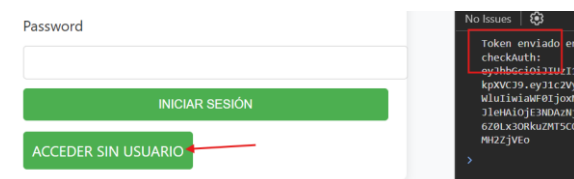
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFTZSI6ImFkbWluIiwiaWF0IjoxNzQwMzYwMjc4LCJleHAiOiJlZmVudG9uLmR5bG9uIn0

3\_



← → ↻

4\_



Password

INICIAR SESIÓN

ACCEDER SIN USUARIO

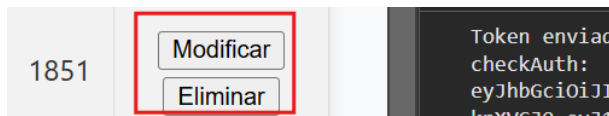
No Issues

Token enviado en

checkAuth:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFTZSI6ImFkbWluIiwiaWF0IjoxNzQwMzYwMjc4LCJleHAiOiJlZmVudG9uLmR5bG9uIn0

5\_



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

Las demás funcionalidades (update, delete y create) se han modificado dentro de las funciones ya existentes, hemos agregado la condición correspondiente a la validación del token, si existe un token podremos modificar los libros, si no encuentra un token aparecerán mensajes de error al dar clic en alguno de los botones. Este caso sucederá en “Acceder sin iniciar sesión” porque lo hemos configurado de tal manera que cumpla con dicha funcionalidad:

## 1\_DELETE

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

```
let response = await fetch(apiUrl, {
  method: "DELETE",
  headers: {
    "Content-Type": "application/json",
    "Authorization": `Bearer ${token}`,
  },
  body: JSON.stringify(deletedBook)
});
```

## 2\_UPDATE

```
async function editBook(event) {
  const token = localStorage.getItem('token');
  if (!token) {
    alert("⚠ Debes iniciar sesión para modificar libros.");
    return;
  }
}
```

```
let response = await fetch(apiUrl, {
  method: "PUT",
  headers: {
    "Content-Type": "application/json",
    'Authorization': `Bearer ${token}`,
  },
  body: JSON.stringify(modifiedBook)
});
```

### 3\_CREATE

```
async function createBook() {
  const token = localStorage.getItem('token');
  if (!token) {
    alert("⚠ Debes iniciar sesión para crear libros.");
    return;
  }
}
```

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