1. ¿Cuáles son las partes de la arquitectura de la aplicación? (con capturas)

La arquitectura de la aplicación se basa en el patrón MVC (Modelo-Vista-Controlador) y está estructurada de la siguiente forma:

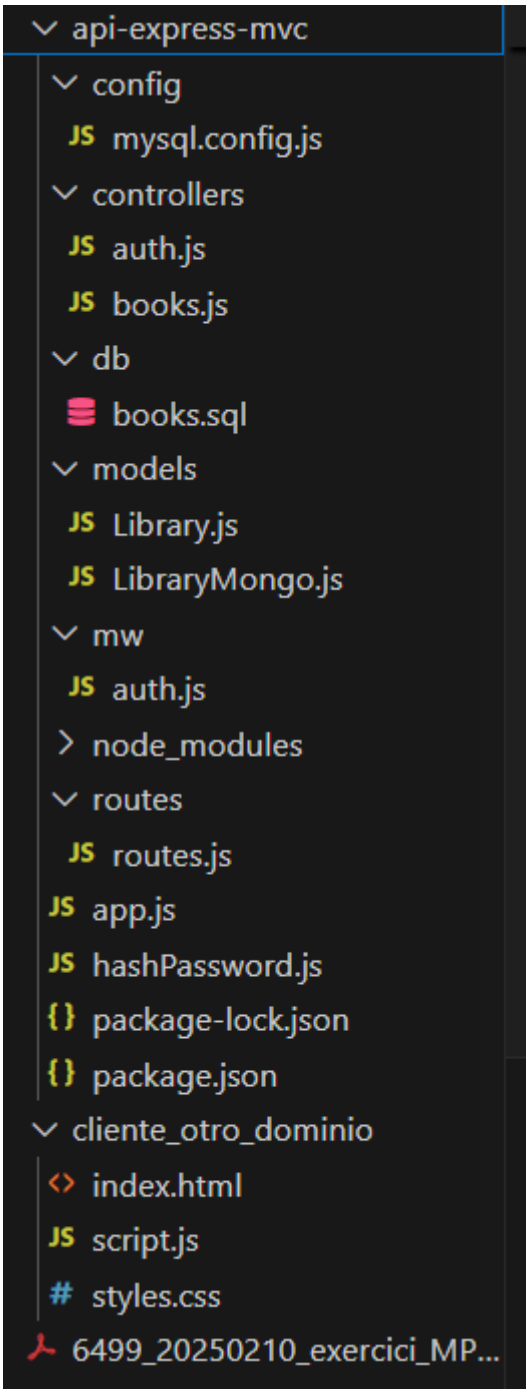
- Modelo (Model): Los modelos son responsables de interactuar con la base de datos y manejar los datos de la aplicación.
Utilizamos los modelos **Usuario.php** y **Anuncio.php**. Aquí es donde se define la lógica de negocio y la estructura de los datos.

- Vista (View): Las vistas están basadas en Twig, un motor de plantillas que permite separar la lógica de presentación del backend. Se incluyen archivos como layout.twig, dashboard.twig, home.twig, crear_anuncio.twig, entre otros. Las vistas se encargan de mostrar los datos a los usuarios de manera dinámica.

- Controlador (Controller): Los controladores como **AnuncioController.php** y **AuthController.php** manejan la lógica de la aplicación. Reciben las solicitudes del usuario, interactuan con los modelos para obtener o modificar los datos y luego devuelven la respuesta adecuada a las vistas.

- Base de Datos: Inicialmente, la aplicación utilizaba MySQL para almacenar los anuncios y los usuarios. Luego, se adaptó a MongoDB, lo que implicó cambios en la estructura y acceso a la base de datos, ya que MongoDB es una base de datos NoSQL y no usa esquemas rígidos como MySQL.

Captura:


api-express-mvc
config
mysql.config.js
controllers
auth.js
books.js
db
books.sql
models
Library.js
LibraryMongo.js
mw
auth.js
node_modules
routes
routes.js
app.js
hashPassword.js
package-lock.json
package.json
cliente_otro_dominio
index.html
script.js
styles.css
6499_20250210_exercici_MP...

2. ¿Cómo se ha hecho la adaptación del modelo de la biblioteca de MySQL a MongoDB?

Para adaptar el modelo de MySQL a MongoDB, fue necesario cambiar la forma en que se estructuran los datos y cómo se accede a ellos. Aquí están los pasos principales que se siguieron:

Cambio de Motor de Base de Datos: Se pasó de MySQL a MongoDB. MongoDB es una base de datos NoSQL que almacena datos en formato BSON (similar a JSON), lo que permite mayor flexibilidad en cuanto a las relaciones entre los datos.

Modelos Adaptados: En lugar de usar tablas, se definieron colecciones en MongoDB. Los modelos como Anuncio.php se adaptaron para trabajar con documentos en lugar de registros.

Manejo de Conexión: Se modificaron los controladores y el modelo para que utilizaran una conexión con MongoDB en lugar de MySQL. Para esto, se configuró el cliente MongoDB en el archivo config.php.

Consultas MongoDB: Las consultas SQL fueron reemplazadas por consultas específicas de MongoDB usando su sintaxis, como .find(), .insertOne(), .updateOne(), y .deleteOne().

Captura:
Interactuar con la DB en MYSQL en Library.js:

```
const mysql = require("mysql2");
const dbConfig = require("../config/mysql.config.js");

class Library {
  constructor() {
    // 1. Declaramos la conexión
    this.connection = mysql.createPool({
      host: dbConfig.HOST,
      user: dbConfig.USER,
      password: dbConfig.PASSWORD,
      database: dbConfig.DB,
      waitForConnections: true,
      connectionLimit: 10,
      queueLimit: 0
    }).promise();
  }

  // Obtener todos los libros
  listAll = async () => {
    try {
      const [results] = await this.connection.query("SELECT * FROM books");
      return results.map(book => ({
        id: book.id,  // Asegurarse de devolver 'id' para ser consistente con MongoDB
        title: book.title,
        author: book.author,
        year: book.year
      }));
    } catch (error) {
      console.error("Error al listar libros:", error);
      throw error;
    }
  };
```

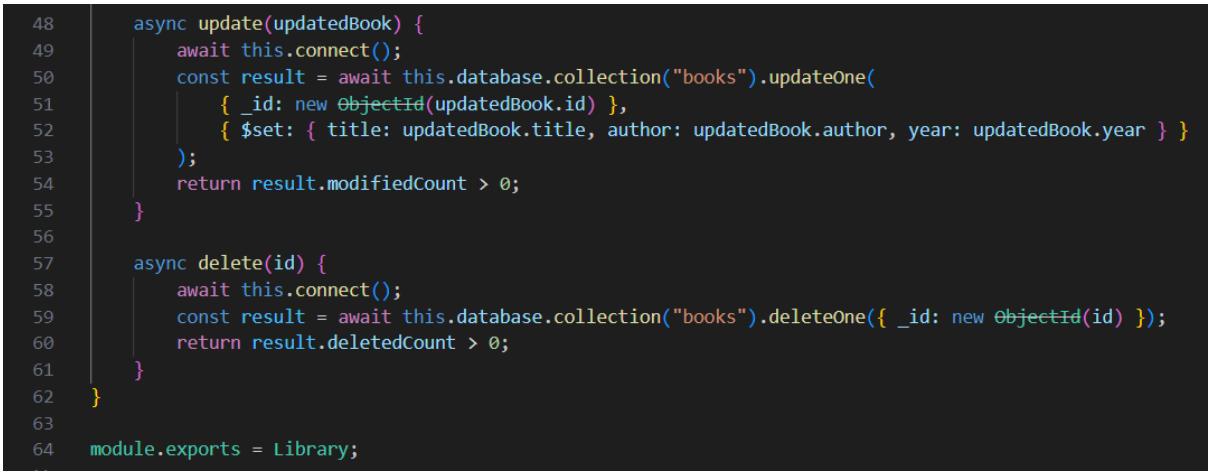
```
34    // Crear un nuevo libro
35    create = async (newBook) => {
36      try {
37        const [result] = await this.connection.query("INSERT INTO books SET ?", newBook);
38        return result.insertId; // Devolver el ID del libro insertado
39      } catch (error) {
40        console.error("Error al crear libro:", error);
41        throw error;
42      }
43    };
44
45    // Actualizar un libro existente
46    update = async (updatedBook) => {
47      try {
48        const { id, title, author, year } = updatedBook;
49        const [result] = await this.connection.query(
50          "UPDATE books SET title = ?, author = ?, year = ? WHERE id = ?",
51          [title, author, year, id]
52        );
53        return result.affectedRows > 0;
54      } catch (error) {
55        console.error("Error al actualizar libro:", error);
56        throw error;
57      }
58    };
59
```

```
60    // Eliminar un libro por ID
61    delete = async (id) => {
62      try {
63        const [result] = await this.connection.query("DELETE FROM books WHERE id = ?", [id]);
64        return result.affectedRows > 0;
65      } catch (error) {
66        console.error("Error al eliminar libro:", error);
67        throw error;
68      }
69    };
70
71    // Método para cerrar la conexión (opcional, ya que se usa un pool)
72    close = async () => {
73      try {
74        await this.connection.end();
75        console.log("Conexión cerrada correctamente.");
76      } catch (error) {
77        console.error("Error al cerrar conexión:", error);
78      }
79    };
80  }
81
82  module.exports = Library;
83
```

Interactuar con la DB en MONGODB en LibraryMongo.js:

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

class Library {
    constructor() {
        this.mongoUrl = "mongodb://localhost:27017"; // URL de conexión
        this.dbName = "library"; // Nombre de la base de datos
        this.client = new MongoClient(this.mongoUrl);
        this.database = null;
    }

    async connect() {
        if (!this.database) {
            await this.client.connect();
            this.database = this.client.db(this.dbName); // Conectar a la base de datos
            console.log(`Conectado a la base de datos: ${this.dbName}`);
        }
    }

    async close() {
        await this.client.close();
        this.database = null;
        console.log("Conexión cerrada");
    }

```

```js
    async close() {
        await this.client.close();
        this.database = null;
        console.log("Conexión cerrada");
    }

    // async listAll() {
    //     await this.connect();
    //     return await this.database.collection("books").find({}).toArray();
    // }

    listAll = async () => {
        await this.connect();
        const books = await this.database.collection("books").find({}).toArray();
        return books.map(book => ({
            id: book._id, // Usa `_id` como `id`
            title: book.title,
            author: book.author,
            year: book.year,
        }));
    };


    async create(newBook) {
        await this.connect();
        const result = await this.database.collection("books").insertOne(newBook);
        return result.insertedId;
    }
```



```
    async update(updatedBook) {
        await this.connect();
        const result = await this.database.collection("books").updateOne(
            { _id: new ObjectId(updatedBook.id) },
            { $set: { title: updatedBook.title, author: updatedBook.author, year: updatedBook.year } }
        );
        return result.modifiedCount > 0;
    }

    async delete(id) {
        await this.connect();
        const result = await this.database.collection("books").deleteOne({ _id: new ObjectId(id) });
        return result.deletedCount > 0;
    }
}

module.exports = Library;
```

3. Capturas de la funcionalidad completa usando MongoDB (listado, adición, modificación y eliminación de libros en la base de datos).

Una vez adaptado el modelo y la base de datos a MongoDB, se implementaron las siguientes funcionalidades:

Listado de Libros (Anuncios): Los anuncios ahora se listan desde MongoDB usando la consulta .find().

Adición de Libros (Anuncios): Se puede agregar un anuncio a la base de datos utilizando .insertOne() o .insertMany().

Modificación de Libros (Anuncios): Los anuncios se pueden actualizar usando .updateOne() para cambiar los campos como el título, descripción, precio, etc.

Eliminación de Libros (Anuncios): Los anuncios se eliminan usando .deleteOne() o .deleteMany() dependiendo del caso.

Captura:
Iniciar conexión con la DB en MYSQL en Library.js

```
const mysql = require("mysql2");
const dbConfig = require("../config/mysql.config.js");

class Library {
  constructor() {
    // 1. Declaramos la conexión
    this.connection = mysql.createPool({
      host: dbConfig.HOST,
      user: dbConfig.USER,
      password: dbConfig.PASSWORD,
      database: dbConfig.DB,
      waitForConnections: true,
      connectionLimit: 10,
      queueLimit: 0
    }).promise();
  }
```

Iniciar conexión con la DB en MONGODB en LibraryMongo.js

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

class Library {
    constructor() {
        this.mongoUrl = "mongodb://localhost:27017"; // URL de conexión
        this.dbName = "library"; // Nombre de la base de datos
        this.client = new MongoClient(this.mongoUrl);
        this.database = null;
    }

    async connect() {
        if (!this.database) {
            await this.client.connect();
            this.database = this.client.db(this.dbName); // Conectar a la base de datos
            console.log(`Conectado a la base de datos: ${this.dbName}`);
        }
    }

    async close() {
        await this.client.close();
        this.database = null;
        console.log("Conexión cerrada");
    }
```

Interactuar con la DB en MYSQL en Library.js:

```
// Crear un nuevo libro
create = async (newBook) => {
  try {
    const [result] = await this.connection.query("INSERT INTO books SET ?", newBook);
    return result.insertId; // Devolver el ID del libro insertado
  } catch (error) {
    console.error("Error al crear libro:", error);
    throw error;
  }
};

// Actualizar un libro existente
update = async (updatedBook) => {
  try {
    const { id, title, author, year } = updatedBook;
    const [result] = await this.connection.query(
      "UPDATE books SET title = ?, author = ?, year = ? WHERE id = ?",
      [title, author, year, id]
    );
    return result.affectedRows > 0;
  } catch (error) {
    console.error("Error al actualizar libro:", error);
    throw error;
  }
};
```

Interactuar con la DB en MONGODB en LibraryMongo.js:

```
    async create(newBook) {
        await this.connect();
        const result = await this.database.collection("books").insertOne(newBook);
        return result.insertedId;
    }

    async update(updatedBook) {
        await this.connect();
        const result = await this.database.collection("books").updateOne(
            { _id: new ObjectId(updatedBook.id) },
            { $set: { title: updatedBook.title, author: updatedBook.author, year: updatedBook.year } }
        );
        return result.modifiedCount > 0;
    }

    async delete(id) {
        await this.connect();
        const result = await this.database.collection("books").deleteOne({ _id: new ObjectId(id) });
        return result.deletedCount > 0;
    }
}
```

4. Explicación de los cambios que se han tenido que hacer (backend y frontend) para implementar la autenticación JWT.

La implementación de la autenticación JWT (JSON Web Token) requirió una serie de cambios tanto en el backend como en el frontend:

Backend:

- Instalación de la librería jsonwebtoken para manejar los tokens JWT.

- Al iniciar sesión, se genera un token con jsonwebtoken.sign(payload, secret, options) y se envía al cliente.

- El token se utiliza en las solicitudes subsiguientes para acceder a recursos protegidos, siendo verificado con jsonwebtoken.verify(token, secret).

- Se modificaron los controladores de autenticación (por ejemplo, AuthController.php) para crear un JWT cuando el usuario inicia sesión correctamente. Este token se genera a partir de la información del usuario y se firma con una clave secreta.

- Los endpoints que requieren autenticación fueron protegidos. Se verificó el token JWT en cada solicitud para asegurar que el usuario esté autenticado antes de acceder a ciertos recursos (como crear o editar anuncios).

Frontend:

- En el frontend, el token se guarda en el almacenamiento local (LocalStorage) usando Authorization: Bearer <token> y se incluye en las cabeceras de las solicitudes mediante el encabezado Authorization.

- Se modificaron las peticiones AJAX para incluir el token JWT en el encabezado Authorization de cada solicitud.

Captura:
Generación y verificación del token JWT

```
const jwt = require('jsonwebtoken');

const SECRET_KEY = "tu_clave_secreta";

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

    if (!token) {
        return res.status(403).json({ message: "Token requerido" });
    }

    try {
        const decoded = jwt.verify(token.split(" ")[1], SECRET_KEY);
        req.userId = decoded.id;
        next();
    } catch (err) {
        return res.status(401).json({ message: "Token inválido" });
    }
};

const generateToken = (userId) => {
    return jwt.sign({ id: userId }, SECRET_KEY, { expiresIn: "2h" });
};

module.exports = { verifyToken, generateToken };

```