**DOCUMENTACIÓN INTEGRACIÓN**

**PONENTES**

**JUAN DAVID SERRANO**

**MIGUEL ANGEL ROMERO**

**LAURA GAONA CASTAÑO**

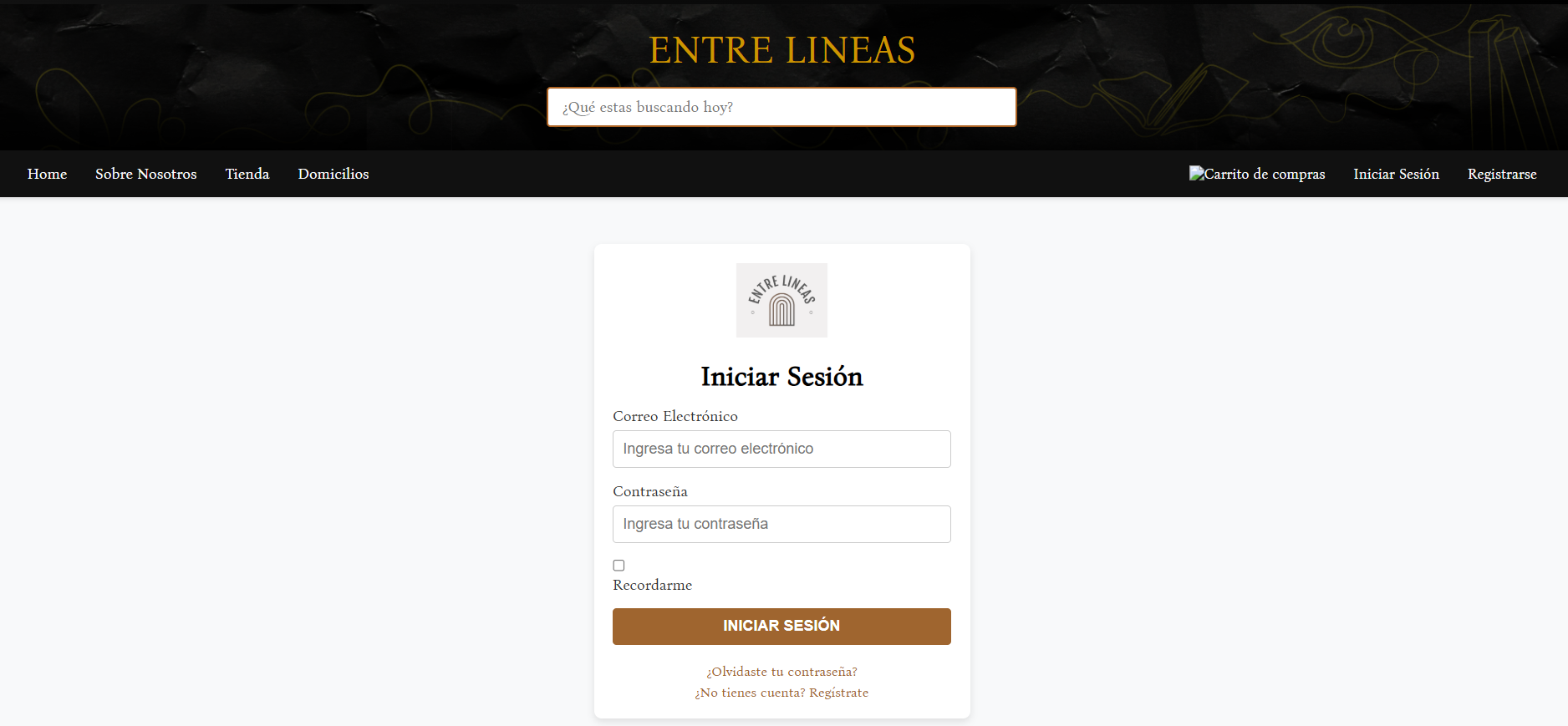
**INGENIERIA DE SOFTWARE**

**UNIVERSIDAD MANUELA BELTRAN**

**2025**

**Documentación**

**Proyecto Entre Líneas**

****

**GUIA N°5,6,7 & 8**

**1. Revisión y definición de los requisitos**

**Requisitos funcionales:**

* El usuario puede **agregar, ver, actualizar y eliminar** libros del carrito.
* El sistema permite **visualizar todos los libros** almacenados en la base de datos.
* El usuario puede **editar los datos** de los libros mediante formularios.

**Requisitos no funcionales:**

* Uso de **PostgreSQL** como base de datos.
* El sistema debe responder en menos de 2 segundos por petición.
* La interfaz debe ser clara, amigable y **responsive**.

**Casos de uso definidos:**

* Registrar libro.
* Editar libro.
* Eliminar libro.
* Visualizar listado de libros.

**2. Desarrollo de la API CRUD con PostgreSQL**

El archivo server.js contiene las operaciones necesarias de Express conectadas a PostgreSQL. Se incluyen rutas para:

* GET /libros: obtener todos los libros.
* POST /libros: agregar un libro.
* PUT /libros/:id: actualizar un libro.
* DELETE /libros/:id: eliminar un libro.

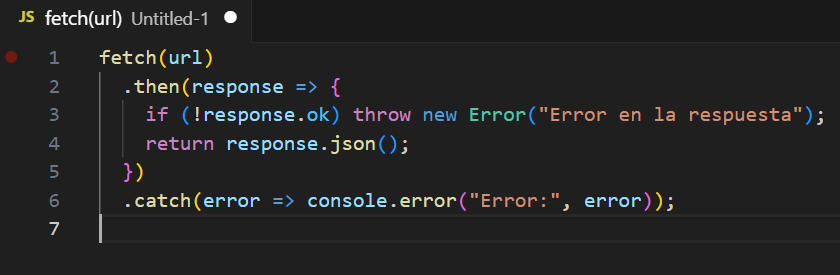
**3. Integración Frontend-Backend**

**Comunicación Frontend ↔ Backend:**

* Uso de fetch () para realizar las peticiones desde app.js.

**Manejo de errores en el frontend:**

* Actualmente: “Código a utilizar”



**Responsividad:**

* El archivo style.css tiene diseño bastante llamativo utilizando una estructura eficaz.

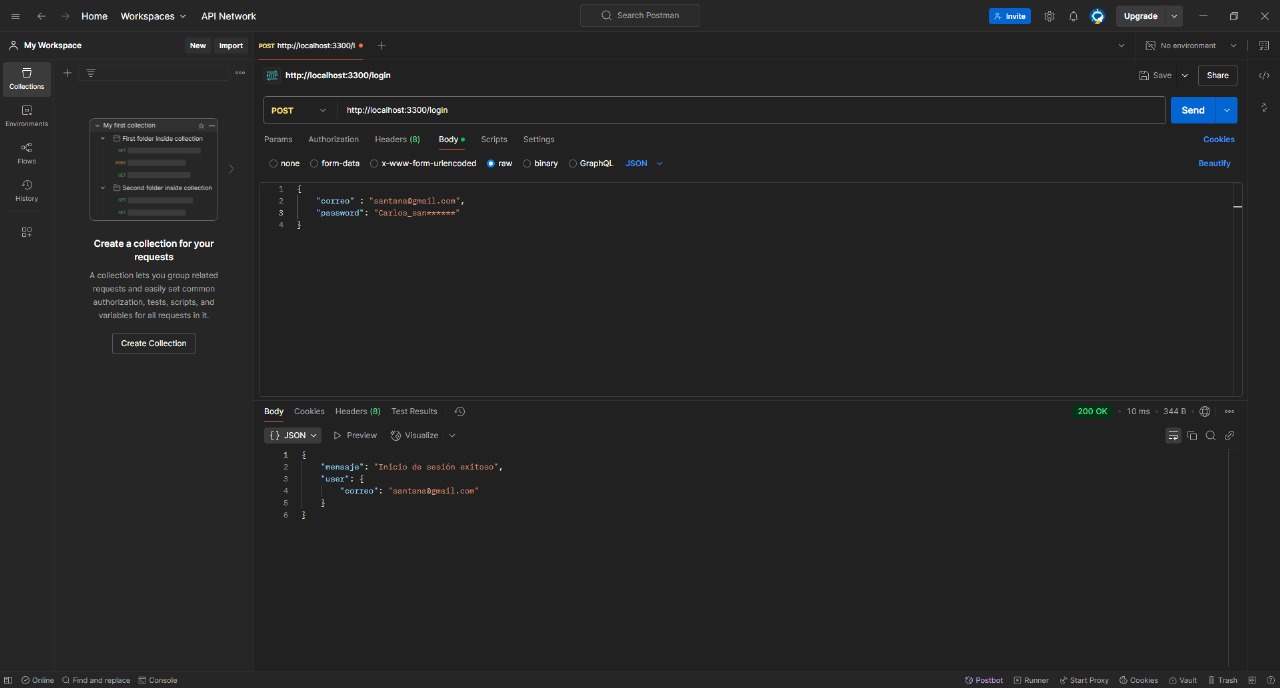
**Actualización de la UI:**

* En la parte del frontend recarga la lista de libros después de agregar, editar o eliminar, lo cual es algo adecuado para la experiencia de usuario dentro de la plataforma.

**4. Pruebas**

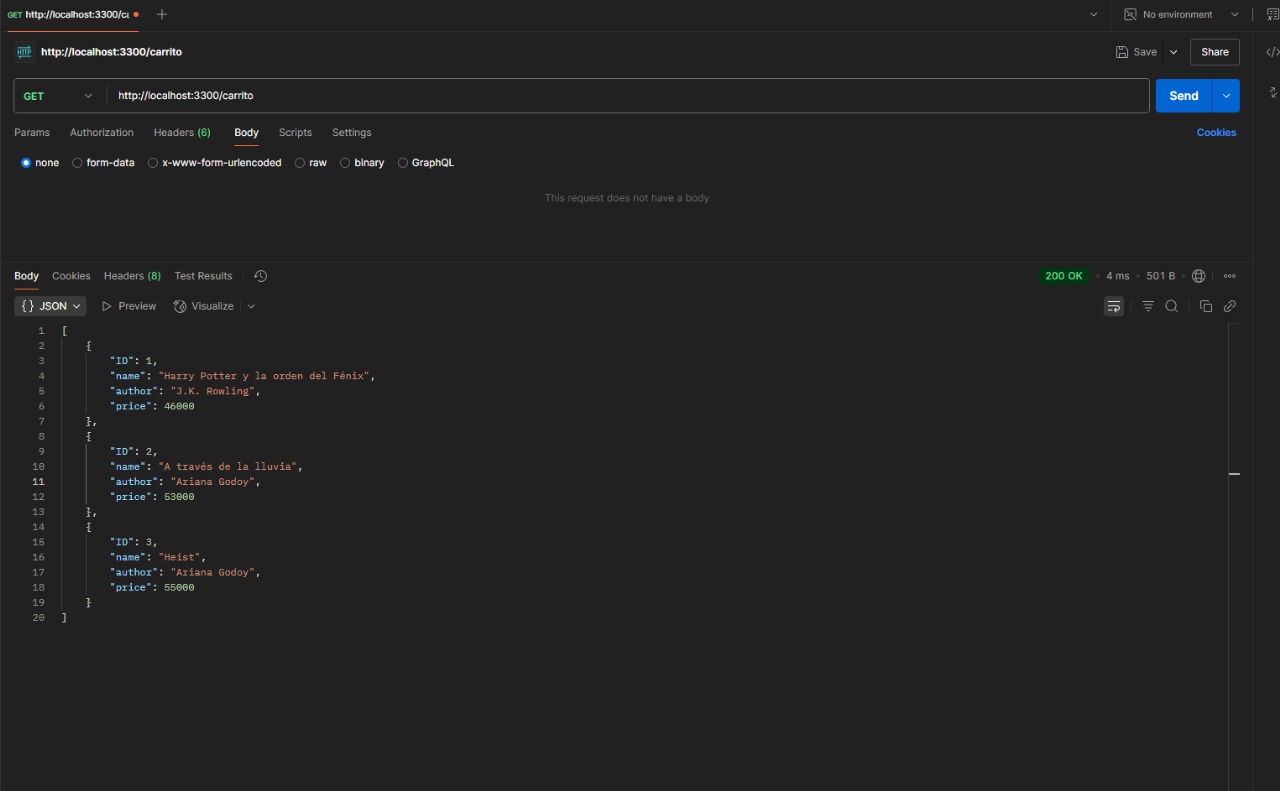
PRUEBAS DE BACKEND

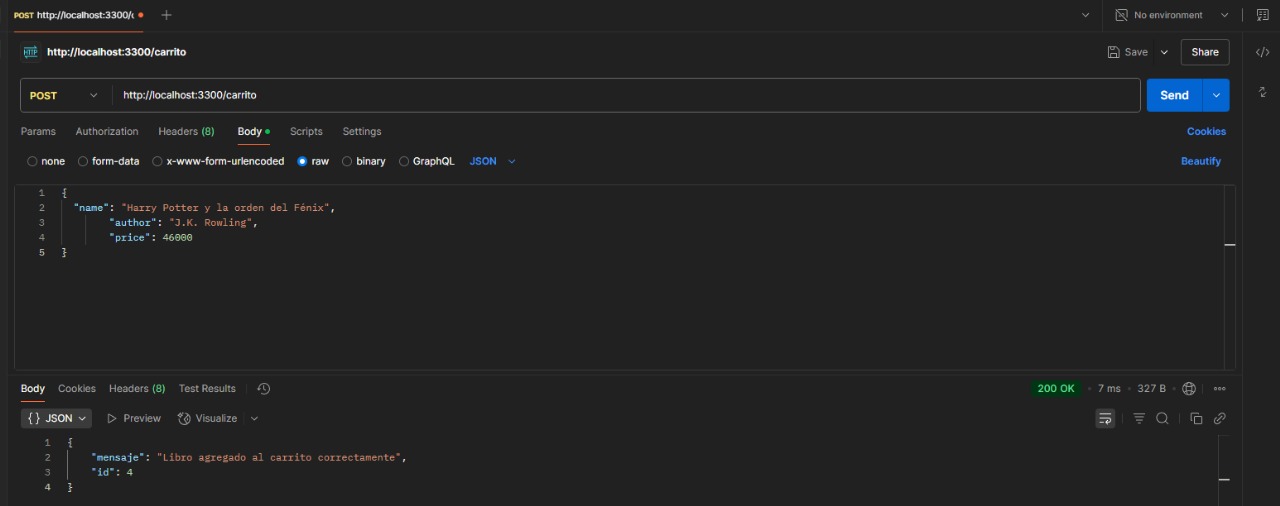
REGISTRO DE USUARIO

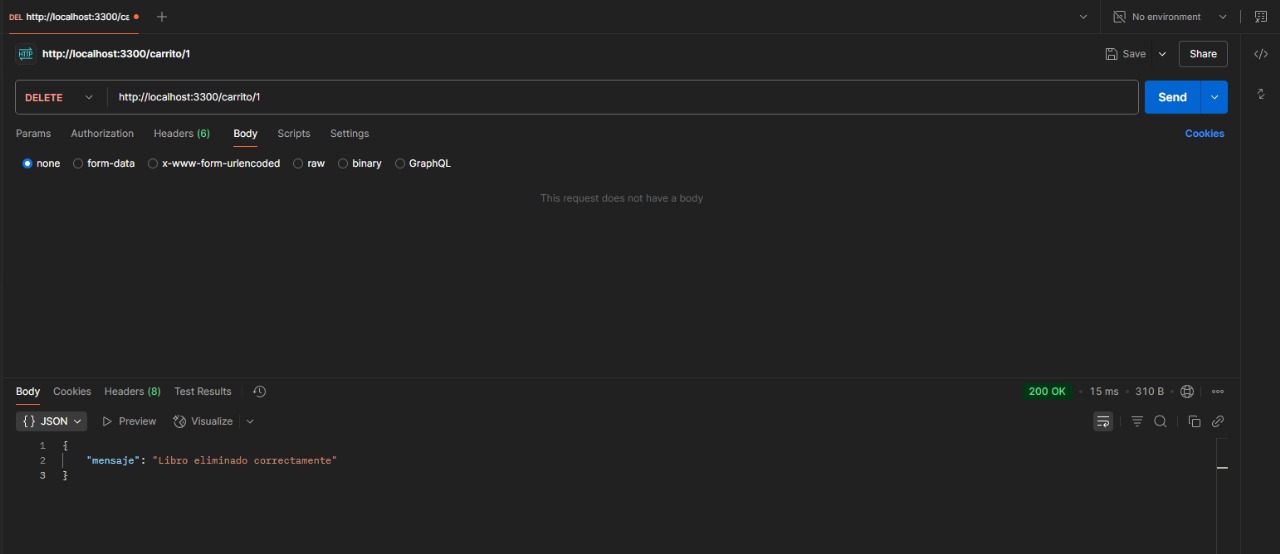
1. RUTAS DE REGISTRO
2. LÓGICA USANDO FECHT. ENVIAMOS LOS DATOS AL BACKEND USANDO FETCH
3. Usamos POSTMAN para verificar que el registro sea exitoso, es decir que haya conexión con la base de datos y el método POST esté funcionando correctamente.
4. PRUEBAS
5. DESPLIEGUE  ****

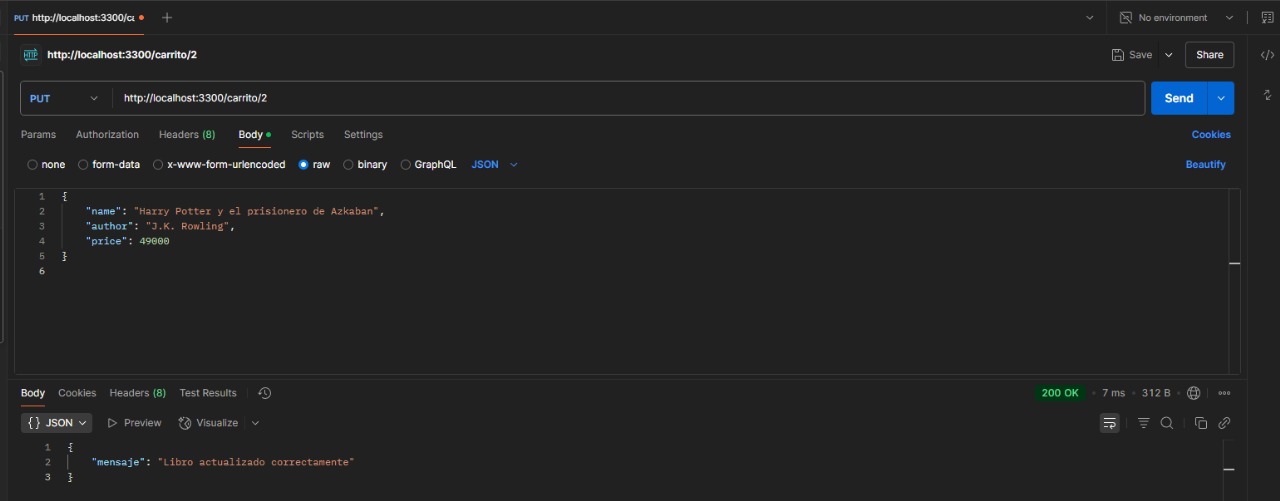
PRUEBAS CARRITO DE COMPRAS

VER LOS LIBROS EN EL CARRITO

****

ELIMINAR ELEMENTOS DEL CARRITO ****

AÑADIR ELEMENTOS AL CARRITO****

MODIFICAR ELEMENTOS DEL CARRITO****

**En el documento ZIP que se tenia en el momento nos reflejaba algunos errores los cuales se documentaron de la siguiente manera.**

ERROR 404-Not Found

Significa que el servidor no encuentra el recurso solicitado.

En este caso no se reflejaba la pagina a la cual sería dirigido, Dentro de la app Entre Líneas, al intentar ingresar a tienda, domicilios y carrito de compras



           HTTP/1.1 404 Not Found

1. Este error ocurre cuando:  
   1.1. La URL está mal escrita.  
   1.2. El archivo o página fue eliminado o movido.  
   1.3. El enlace está roto.

ERRORES DE GET

GET se usa para solicitar datos desde un servidor.

1. Puede verse reflejado el error cuando:  
   1. URL mal escrita.  
   2. Parámetros faltantes en la URL.  
   3. Falta de permisos (puede devolver error 403 o 401).  
   4. Recurso no existe (error 404).

**Fases que se utilizaron para la integración del proyecto**

**Fase n°1 Revisión y definición de requisitos**

**Requisitos Funcionales:**

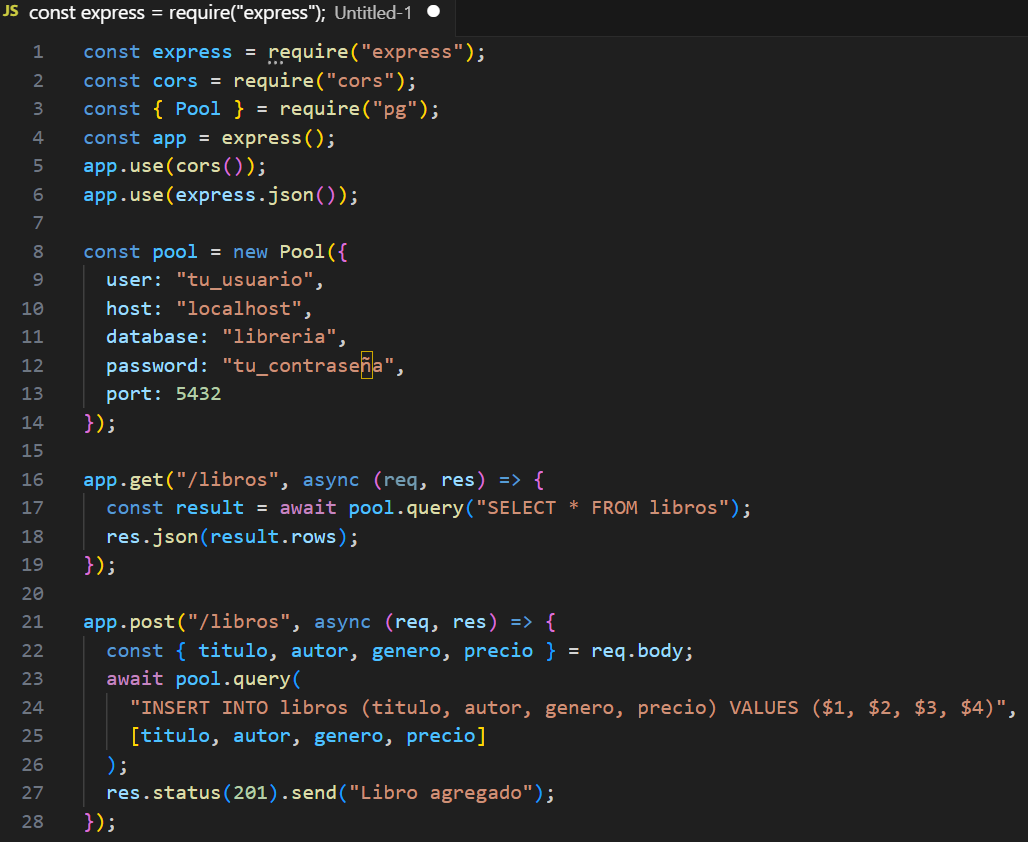
* Registro e inicio de sesión de usuarios.
* Consulta y filtrado de libros.
* Gestión de carrito de compras.
* Procesamiento de pagos simulado.
* Reseñas de libros.
* Panel administrativo para CRUD de libros.

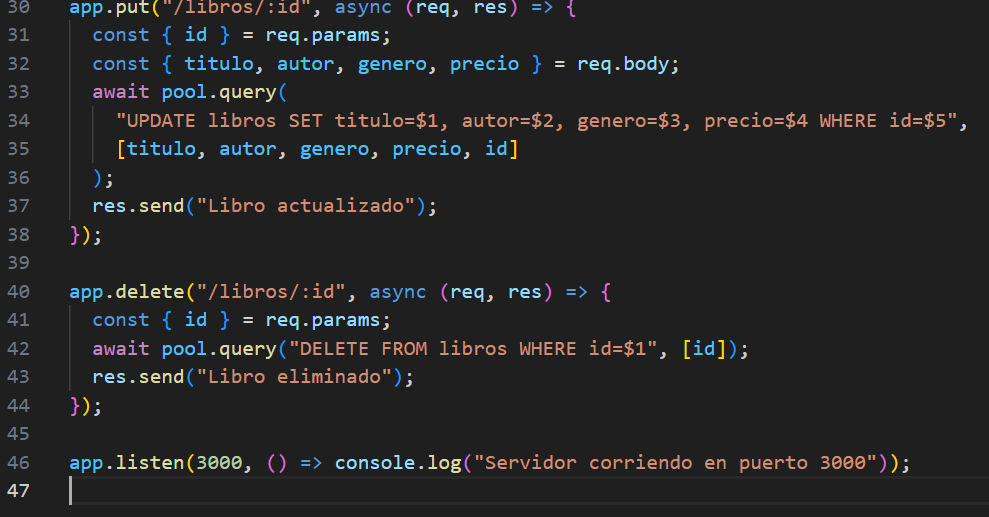
**Requisitos No Funcionales:**

* Interfaz responsive.
* Comunicación asincrónica usando fetch.
* Base de datos relacional (MySQL actualmente, se puede adaptar a PostgreSQL).
* Control de versiones con Git y despliegue futuro en servidor personal.

**Fase 2: Desarrollo de la API CRUD con PostgreSQL**

API con Express y PostgreSQL:

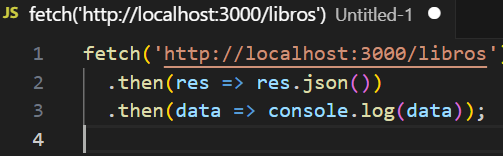




**Fase 3: Integración y pruebas**

**Comunicación Frontend ↔ Backend:**

* Uso del fetch() en JS:



**Errores gestionados en el documento:**

* En la validación del campo de backend se hallaron (campos vacíos y duplicados).
* try/catch en servidor y mensajes claros al cliente.

**Rutas API documentadas:**

| **Método** | **Ruta** | **Descripción** |
| --- | --- | --- |
| GET | /libros | Lista todos los libros |
| POST | /libros | Agrega un nuevo libro |
| PUT | /libros/:id | Edita un libro |
| DELETE | /libros/:id | Elimina un libro |

**Pruebas en Postman (Modelo)**

**1. GET /libros**

* Esperado: Código 200, lista de libros.

**2. POST /libros**

* JSON enviado:

{ "titulo": "1984", "autor": "George Orwell", "genero": "Distopía", "precio": 35.5 }

* Esperado: Código 201 Created.

**3. PUT /libros/1**

* Actualizado el título.
* Esperado: Código 200 OK.

**4. DELETE /libros/1**

* Esperado: Código 200 OK.

**DESPLIEGUE**

Se utilizo Linux para el despliegue del proyecto junto a la base de datos incorporada en las que nos permite visualizar en campo de los diferentes métodos utilizados; además de permitirnos incorporar la integración entre el front y el back.

**Requisitos que se tuvieron en cuenta:**

**1. Dependencias generales**

* Node.js
* PostgreSQL
* Navegador web
* Git

**2. Archivos y carpetas necesarias**

* index.html, style.css, app.js → Frontend
* server.js → Backend (Express)
* Libreria\_BD.sql → Script para crear la base de datos
* package.json y package-lock.json → Dependencias del backend
* README.md → Instrucciones para correr el proyecto

**Pasos para tener en cuenta:**

* **Backend**

1. Crear base de datos en PostgreSQL o usar MySQL.
2. Importar Libreria\_BD.sql a la base de datos.
3. Crear archivo .env (si usas variables de entorno).
4. Instalar dependencias:

npm install express pg cors dotenv

1. Ejecutar el servidor:

node server.js

* **Frontend**

1. Abrir index.html en el navegador o levantar un servidor local con Live Server.
2. Asegurarse de que las peticiones fetch se conecten a <http://localhost:8000>.

**PREGUNTAS SOLICITADAS❓**

* 1. **INTEGRACIÓN FRONTEND-BACKEND**

**¿Cómo haces que el frontend se comunique con el backend (Express y PostgreSQL)?**

Usamos la función fetch() en JavaScript para enviar y recibir peticiones HTTP desde el frontend al backend. Estas peticiones llegan a rutas definidas en Express, las cuales interactúan con la base de datos PostgreSQL mediante el paquete pg.

**¿Qué métodos usas para realizar peticiones HTTP? ¿Por qué elegiste esta opción?**

Usamos fetch() porque está integrado en JavaScript y permite hacer peticiones asincrónicas fácilmente sin necesidad de instalar bibliotecas externas. Para este tipo de proyecto básico, es suficiente y eficiente.

**¿Cómo gestionas los errores que pueden surgir en el frontend al interactuar con la API?**

Usamos try/catch y validaciones para controlar errores como respuestas no válidas, campos vacíos o errores del servidor. En el frontend se muestra un mensaje de error si falla la operación, y se evita que el sistema se congele.

**¿Cómo aseguras que el frontend sea responsivo y fácil de usar en diferentes dispositivos?**

Usamos HTML y CSS con un diseño adaptativo (responsive) pensado en móviles y pantallas de escritorio. Aunque básico, se sigue una estructura con contenedores flexibles y media queries en los archivos CSS.

**¿Cómo actualizas la interfaz de usuario cuando los datos cambian?**

Luego de cada operación CRUD (crear, editar, eliminar), hacemos una nueva petición fetch() para obtener los datos actualizados y renderizarlos en el DOM (HTML). Esto mantiene la vista sincronizada con la base de datos.

* 1. **PRUEBAS**

**Tipos de pruebas**

* **Unitarias:** Registro, login, manipulación de datos de libros.
* **Integración:** Flujo completo frontend → backend → base de datos.

**Pruebas CRUD con Postman**

* **POST /libros**: Se envia JSON con los datos del libro. Se espera 201 Created.
* **GET /libros**: Se espera un array de libros y 200 OK.
* **PUT /libros/:id**: Se edita un libro existente. Se espera 200 OK.
* **DELETE /libros/:id**: Se elimina un libro. Se espera 200 OK.

**Verificación de integración**

* Las acciones en el frontend reflejan cambios reales en la base de datos.
* La comunicación fetch() se confirma con respuestas correctas.

**Manejo de errores comunes**

* Validaciones de formularios en frontend.
* Mensajes de error en backend cuando hay datos inválidos o faltantes.

**¿Qué tipo de pruebas realizaste para asegurarte de que la API funcione correctamente?**

Se realizaron pruebas **unitarias** y **de integración**:

* Unitarias: Validación de registro de usuarios, edición de libros, eliminación.
* Integración: Flujo completo desde frontend hasta base de datos usando rutas reales con datos válidos e inválidos.

**¿Cómo probaste las funciones CRUD en Postman o Insomnia? ¿Qué resultados esperabas?**

Usamos Postman. Enviamos peticiones con datos JSON a cada ruta:

* POST /libros: Esperábamos 201 Created
* GET /libros: Esperábamos 200 OK y listado de libros
* PUT /libros/:id: Esperábamos 200 OK al editar
* DELETE /libros/:id: Esperábamos 200 OK al borrar

Todas devolvieron respuestas correctas, validando el funcionamiento.

**¿Cómo verificaste que el frontend interactúa correctamente con el backend?**

Usamos tanto Postman como la interfaz del navegador. Al realizar una operación en la interfaz (como agregar un libro), revisamos que:

* Se actualizara en la base de datos.
* Los datos se mostraran de nuevo en pantalla correctamente.
* El backend devolviera el status y JSON esperado.

**¿Qué harías si se encuentran errores de integración entre el frontend y el backend?**

Primero, revisamos la consola del navegador y del backend para encontrar el error. Luego verificamos:

* Que las rutas coincidan (frontend/backend).
* Que los formatos (JSON, headers) estén correctos.
* Si no se resuelve, nos comunicamos con los otros equipos (Frontend o Backend) para encontrar la solución en conjunto.

**¿Cómo manejas los errores comunes durante las pruebas (por ejemplo, registros no encontrados, datos inválidos)?**

Implementamos validaciones del lado del backend y mensajes de error descriptivos. En el frontend validamos campos antes de enviar peticiones. Si el backend responde con un error, mostramos mensajes adecuados para el usuario final.

GUIA N°6

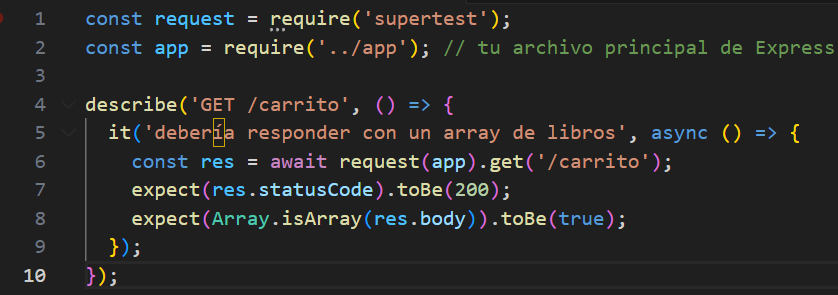
**Desarrollo**

* + **POST /carrito**: Agregar un libro al carrito.
  + **GET /carrito**: Listar todos los libros en el carrito.
  + **PUT /carrito/:id**: Editar la cantidad o información de un libro en el carrito.
  + **DELETE /carrito/:id**: Eliminar un libro del carrito.
* Conexión de la API con PostgreSQL mediante librerías como pg en Node.js.
* Pruebas de funcionamiento usando herramientas como Postman o Insomnia.

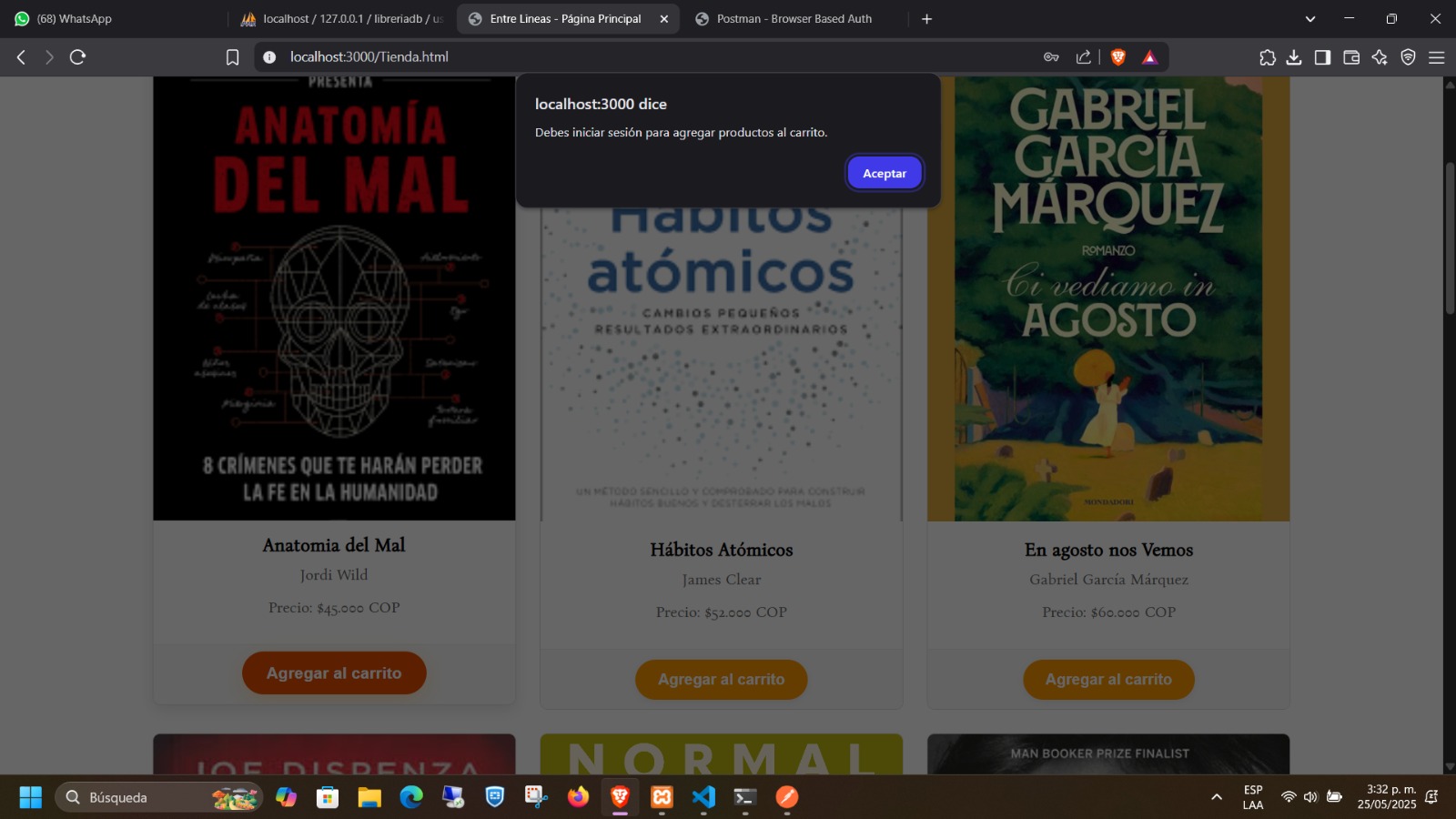
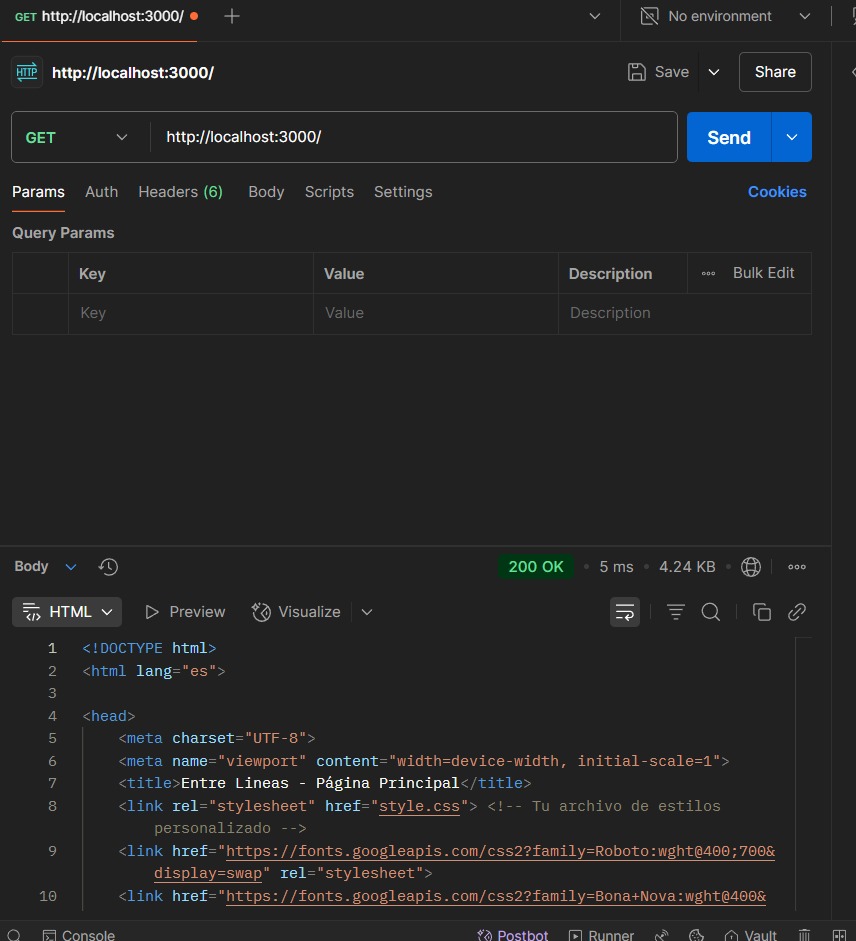
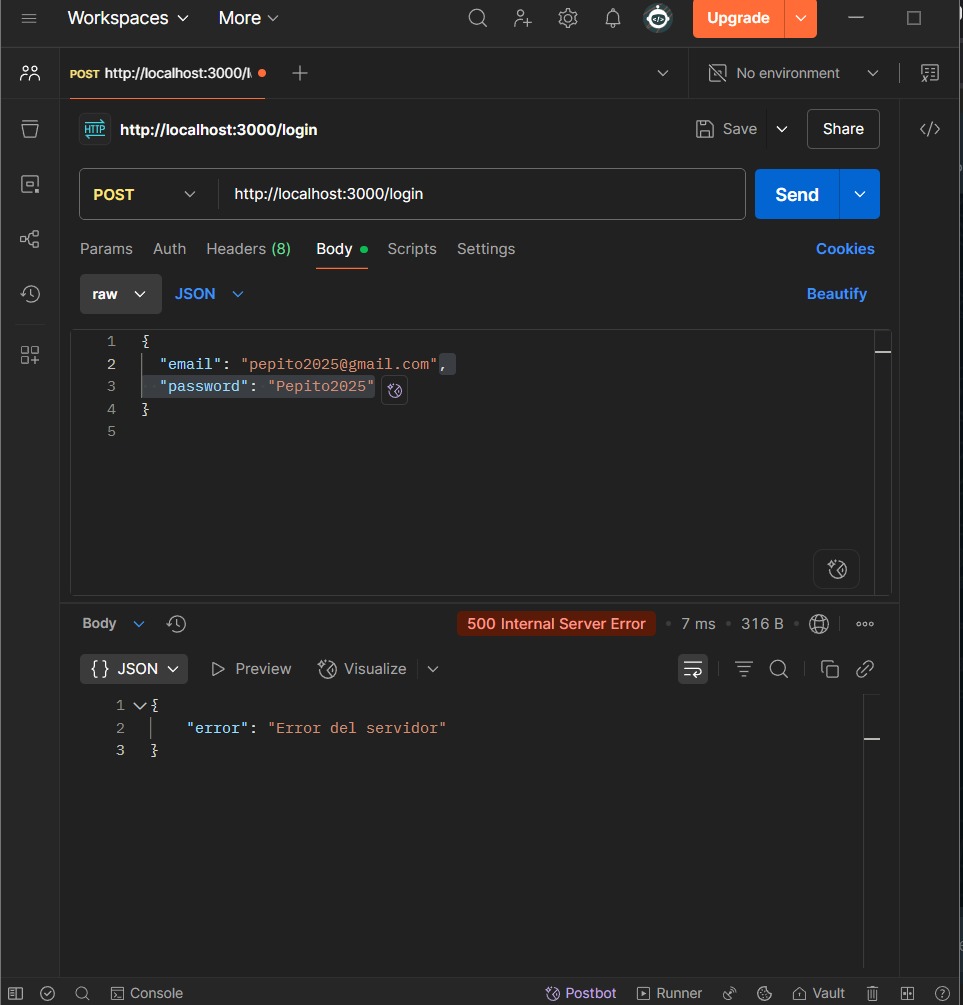
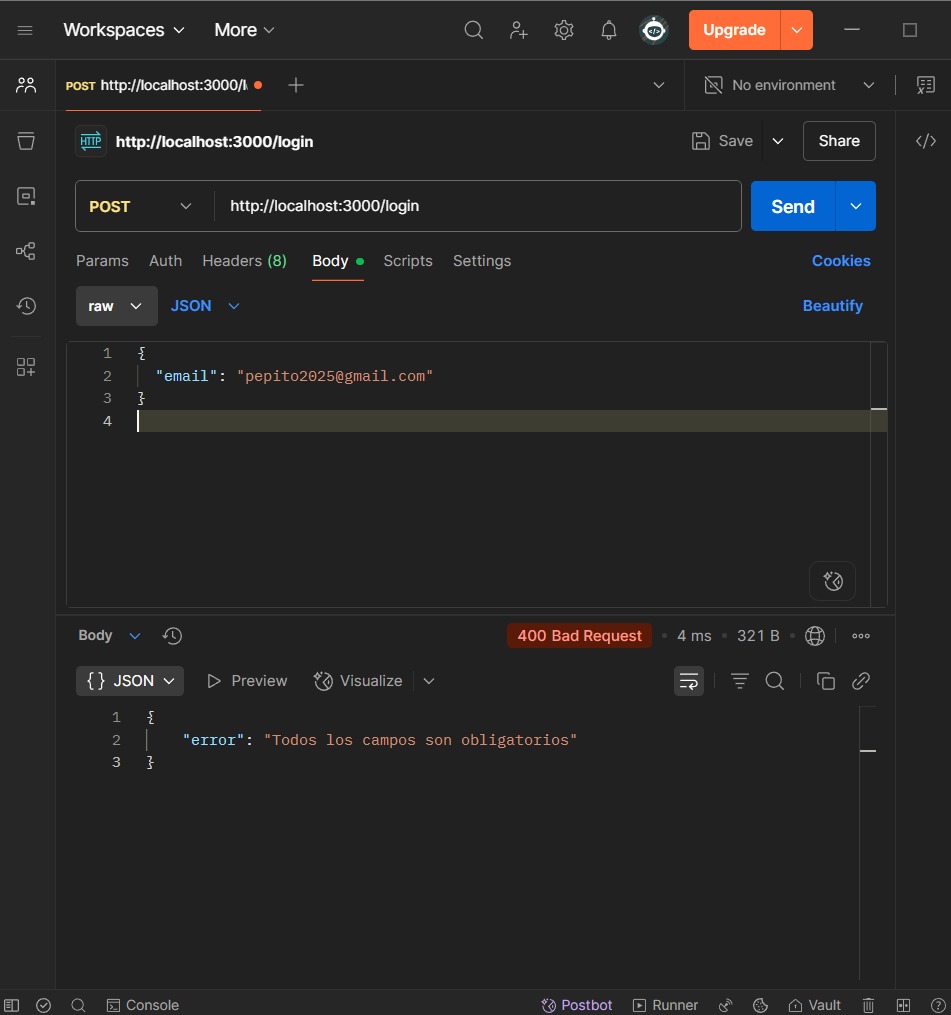
**Pruebas Unitarias**

* Las pruebas unitarias permiten asegurar el correcto funcionamiento de funciones específicas del backend (como los controladores de la API o funciones de acceso a la base de datos).
* En Node.js

**Prueba con Jest:**



Estas pruebas aseguran que el backend continúe funcionando correctamente.



**¿Qué función realiza localStorage?**

localStorage se utiliza en el proyecto en los espacios de login.js, carrito.js, soporte.js, y tienda.js para:

* Guardar información persistente del usuario (como si ha iniciado sesión o el contenido del carrito).
* Mantener datos accesibles incluso después de recargar la página.
* Evitar llamadas innecesarias al servidor.

**¿De dónde obtiene los datos?**

El sistema obtiene datos de:

* **El backend en Express + MySQL**, usando paquetes como mysql, body-parser, y cors.
* **localStorage**, que guarda y recupera datos del navegador.
* **Entradas del usuario**, como formularios de inicio de sesión o acciones dentro del frontend.

**¿Cómo mantener la sesión iniciada?**

El sistema:

1. Guarda un identificador (como idUsuario) o token en localStorage al iniciar sesión.
2. Verifica este dato en cada carga de página para redirigir al usuario según si está logueado o no.
3. No hay un sistema de autenticación robusto (como JWT) mencionado, por lo que depende del localStorage.

**¿Importancia de tener archivos en carpetas separadas (front/back)?**

Separar el **frontend** y **backend** permite:

* Organización clara del código.
* Trabajo independiente en cada parte del sistema.
* Facilita el despliegue y pruebas.
* Mejora la seguridad al evitar exponer código backend al cliente.

**¿Cómo ajustar el JSON?**

Para ajustar el JSON (package.json), puedes:

* Agregar scripts personalizados para correr el servidor:

"scripts": {

"start": "node index.js",

"dev": "nodemon index.js"

}

* Verificar versiones específicas para evitar incompatibilidades:

"dependencies": {

"express": "^5.1.0",

"mysql": "^2.18.1"

}

**¿Cómo configurar el package.json para desplegar en Linux?**

Pasos claves que se tuvieron en cuenta:

1. Hay que asegurarte de que el servidor arranca con node index.js o el archivo principal definido.
2. Agrega start en los scripts.
3. Instala las dependencias en el servidor Linux con:

npm install

1. Ejecuta con:

npm start

1. Puedes usar pm2 para mantener el proceso activo:

npm install -g pm2

pm2 start index.js --name entre\_lineas