PRÁCTICA 6.05 CRUD con `fetch`

Normas de entrega

- En cuanto al **código**:
  - en la **presentación interna**, importan los **comentarios**, la claridad del código, la significación de los nombres elegidos; todo esto debe permitir considerar al programa como **autodocumentado**. No será necesario explicar que es un `if` un `for` pero sí su funcionalidad. Hay que comentar las cosas destacadas y, sobre todo, las **funciones** y **clases** empleadas. La ausencia de comentarios será penalizada,
  - en la **presentación externa**, importan las leyendas aclaratorias, información en pantalla y avisos de ejecución que favorezcan el uso de la aplicación,
  - para el nombre de **variables**, **constantes** y **funciones** se utilizará *lowerCamelCase,*
  - el nombre de los componentes debe comenzar con letra **mayúscula**.
  - la **componentización** debe ser lógica y estar adaptada a las necesidades de la aplicación,
  - todos los formularios en `React` deben ser del tipo **controlados** (a través del **estado**). En caso contrario debe ser debidamente justificado,
  - toda comunicación con el exterior del `DOM` debe ser realizada con funciones asíncronas que dispongan de control de errores (`try{} catch{}`) para gestionar posibles fallos de red o respuestas `HTTP` no exitosas (comprobando la propiedad `response.ok`),
  - se debe implementar el uso de `async/await` para la comunicación asíncrona y
  - los componentes deben consumir los datos a través de un **contexto** de `React` para evitar la propagación manual de `props`.
- En cuanto a la **entrega** de los archivos que componen los ejercicios de `React`:
  - entrega **la práctica** en un sólo proyecto (el nombre a tu discreción),
  - los componentes creados deben estar separados en carpetas (los creados en el Ejercicio1 dentro de una carpeta denominada `Ejercicio1`),
  - el código contendrá ejemplos de ejecución, si procede,
  - comprime la carpeta `src` junto con el fichero `package.json` en un fichero `ZIP` , y
  - sube a **Aules** el fichero comprimido.

Ejercicio 1 - Explotando una API

Retoma el ejercicio "Mi colección de discos" que realizaste en **React** y añádele el código necesario para implementar las siguientes funcionalidades, entre ellas, sustituir el uso de **localStorage** por una **API** externa.

Parte I. Creación de la **API** en local siguiendo estos pasos:

- instala **json-server** en tu proyecto con el comando **npm install -g json-server**,
- crea un archivo **JSON** con la estructura de los discos (asegúrate de que la clave **discos** sea la propiedad raíz),
- ejecuta el servidor con el comando **json-server –watch discos.json –port 3000** (o mejor, crea un comando en la sección **scripts** en el fichero **package.json** de tu proyecto),
- la **API** estará disponible a través de **http://localhost:3000/discos**.

Parte II. Diseño de un contexto de datos (**useContext**). Centraliza la lógica de los discos en un contexto global para que cualquier componente pueda acceder a la información sin intermediarios. Este proveedor debe albergar el estado **discos** y todas las funciones de comunicación con la **API** (leer, guardar, eliminar y actualizar).

Parte III. Carga inicial (**GET**): al cargar la aplicación, utiliza una función asíncrona con **fetch** para obtener el listado de discos.

Se debe controlar el estado de carga mientras la promesa está en estado **pending** (a través de un estado **cargando** y un **componente**).

Parte IV. Guardado (**POST**): tras validar el formulario (según los criterios del ejercicio anterior), los datos se enviarán a la **API** mediante una petición **POST**. El cuerpo de la petición debe ser un objeto **JSON** convertido con **JSON.stringify()**.

Parte V. Actualización de discos (**PUT/PATCH**): se debe añadir la funcionalidad para editar los datos de un disco existente en la colección. Al pulsar el botón Editar (que deberás crear) en un disco, sus datos deben cargar de nuevo en un formulario controlado (lo más lógico es utilizar rutas dinámicas como **/discos/:id**). Al pulsar sobre el botón Actualizar Datos del formulario se realizará una petición **PUT** o **PATCH** para actualizar los datos.

Parte VI. Refactorización con **hooks** personalizados (**custom hooks**). Para mejorar la legibilidad y reutilización del código, encapsula la lógica de las peticiones a la **API** en un **hook** personalizado (**useAPI.js**). Este **hook** recibirá los parámetros necesarios y devolverá los datos obtenidos, el estado que indica la situación de la solicitud y las funciones necesarias para realizar las operaciones **CRUD**. Asegúrate de que el **hook** gestione su propio estado interno de forma aislada, devolviendo únicamente lo necesario.

Crea un nuevo **hook** personalizado (**useDiscos.js**) para consumir el contexto por parte de los componentes hijos. Este **hook** se utilizará para acceder a los datos y funciones compartidas por el contexto.