Ejercicios de Repaso - JavaScript

Esta relación de ejercicios está diseñada para practicar los conceptos fundamentales de JavaScript relacionados con arrays, arrays de objetos, Map, Set, LocalStorage y funciones. Los ejercicios tienen un nivel medio-alto y requieren aplicar varios conceptos de manera conjunta.

Las salidas se mostrarán por consola o como máximo en etiquetas `<p>`.

Para resolver estos ejercicios, utiliza los datos proporcionados en el archivo `src/db/data.js`, que contiene diferentes conjuntos de datos que servirán como base para las prácticas.

Ejercicios de Arrays y Arrays de Objetos

Ejercicio 1: Filtrado y Transformación de Usuarios

Objetivo: Practicar el uso de métodos de arrays como `filter`, `map` y `sort`.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `obtenerUsuariosActivosOrdenados` que:

1. Filtre solo los usuarios activos del array `usuarios`
2. Ordene los usuarios resultantes por edad de forma descendente
3. Transforme cada usuario en un nuevo objeto que solo contenga las propiedades: `nombre`, `email` y `edad`
4. Devuelva el array resultante

Crea funciones adicionales para:

- Ordenar los resultados por edad (ascendente/descendente)
- Buscar por nombre o email
- Mostrar/ocultar usuarios inactivos

Muestra los resultados por consola.

Ejercicio 2: Análisis de Productos por Categoría

Objetivo: Practicar el uso de `reduce` para agrupar datos y realizar cálculos.

Datos necesarios: Array `productos` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `analizarProductosPorCategoria` que:

1. Agrupe los productos del array `productos` por categoría
2. Para cada categoría, calcule:
    - Número total de productos
    - Precio medio de los productos
    - Stock total
    - Valoración media
3. Devuelva un objeto donde las claves sean las categorías y los valores sean objetos con la información calculada

Crea funciones adicionales para:

- Filtrar categorías con stock bajo o valoración alta
- Ordenar categorías por diferentes criterios (productos, precio, stock)
- Identificar categorías con problemas (stock bajo, valoración baja)

Muestra los resultados por consola.

Ejercicio 3: Búsqueda Avanzada de Usuarios

Objetivo: Practicar búsquedas complejas en arrays de objetos.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `buscarUsuariosPorHobbies` que:

1. Acepte como parámetro un array de hobbies a buscar
2. Devuelva todos los usuarios que tengan AL MENOS UNO de los hobbies especificados
3. Ordene los resultados por número de hobbies coincidentes (de más a menos)
4. En caso de empate, ordene alfabéticamente por nombre

Crea funciones adicionales para:

- Contar el número de coincidencias por usuario
- Filtrar usuarios por ciudad o nivel
- Calcular estadísticas de los hobbies más populares

Muestra los resultados por consola.

Ejercicio 4: Cálculo del Total de Pedidos

Objetivo: Practicar el manejo de arrays anidados y cálculos complejos.

Datos necesarios: Array `pedidos` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `calcularTotalPedidos` que:

1. Calcule el total de cada pedido (cantidad × precio unitario para cada producto)
2. Sume los totales de todos los productos de cada pedido
3. Devuelva un array de objetos con la siguiente estructura:

```
{
  idPedido: número,
  total: número,
  numeroProductos: número,
  idUsuario: número,
  estado: string,
  fecha: Date
}
```

Crea funciones adicionales para:

- Calcular comisiones por método de pago
- Obtener estadísticas de ventas por mes
- Filtrar pedidos por estado y rango de fechas

Muestra los resultados por consola.

Ejercicio 5: Estadísticas de Usuarios

Objetivo: Practicar el uso de métodos de arrays para obtener estadísticas.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `obtenerEstadisticasUsuarios` que:

1. Calcule la edad media de todos los usuarios
2. Encuentre el usuario más joven y el más mayor
3. Determine cuántos usuarios hay por ciudad
4. Calcule el porcentaje de usuarios activos
5. Devuelva un objeto con toda esta información

Crea funciones adicionales para:

- Filtrar estadísticas por nivel de usuario
- Comparar estadísticas entre diferentes grupos
- Obtener usuarios destacados por diferentes criterios

Muestra los resultados por consola.

Ejercicios de Map y Set

Ejercicio 6: Gestión de Inventario con Map

Objetivo: Practicar el uso de Map para gestionar inventario.

Datos necesarios: Array `productos` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `gestionarInventario` que:

1. Cree un Map a partir del array `productos` donde la clave sea el id del producto
2. Implemente las siguientes operaciones sobre el Map:
   - `actualizarStock(idProducto, nuevaCantidad)`: Actualiza el stock de un producto
   - `obtenerProductoPorId(idProducto)`: Devuelve el producto con ese id
   - `productosConBajoStock(limite)`: Devuelve un array con productos cuyo stock sea inferior al límite
3. Devuelva un objeto con estas tres funciones y el Map inicial

Crea funciones adicionales para:

- Generar alertas automáticas cuando el stock baje del límite
- Buscar productos por nombre o categoría
- Guardar cambios en LocalStorage

Muestra los resultados por consola.

Ejercicio 7: Sistema de Etiquetas con Set

Objetivo: Practicar el uso de Set para gestionar etiquetas únicas.

Datos necesarios: Array `productos` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `sistemaEtiquetas` que:

1. Extraiga todas las etiquetas del array `productos` y las guarde en un Set para eliminar duplicados

2. Implemente las siguientes operaciones:
    - `agregarEtiqueta(etiqueta)`: Añade una nueva etiqueta al Set
    - `eliminarEtiqueta(etiqueta)`: Elimina una etiqueta del Set
    - `obtenerProductosConEtiqueta(etiqueta)`: Devuelve los productos que contengan esa etiqueta
    - `etiquetasDisponibles()`: Devuelve un array con todas las etiquetas disponibles
3. Devuelva un objeto con el Set de etiquetas y las funciones para gestionarlo

Crea funciones adicionales para:

- Filtrar productos por múltiples etiquetas (AND/OR)
- Contar el número de productos por etiqueta
- Identificar las etiquetas más populares

Muestra los resultados por consola.

Ejercicio 8: Comparación de Colecciones

Objetivo: Practicar operaciones entre Map y Set.

Datos necesarios: Sets `coloresPrimarios`, `coloresSecundarios` y Map `ciudadesPoblacion` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `compararColecciones` que:

1. A partir de los Sets `coloresPrimarios` y `coloresSecundarios`, cree:
    - Un Set con todos los colores (unión)
    - Un Set con los colores que están en ambos (intersección)
    - Un Set con los colores que están en primarios pero no en secundarios (diferencia)
2. A partir del Map `ciudadesPoblacion`, cree otro Map solo con las ciudades cuya población sea superior a 1 millón
3. Devuelva un objeto con todos estos resultados

Crea funciones adicionales para:

- Añadir nuevos elementos a las colecciones
- Calcular estadísticas de las operaciones
- Exportar los resultados de las operaciones

Muestra los resultados por consola.

Ejercicio 9: Cache de Resultados con Map

Objetivo: Practicar el uso de Map como sistema de cache.

Datos necesarios: Array `numeros` del archivo `src/db/data.js` para demostración

Enunciado: Implementa una función llamada `crearCacheFunciones` que:

1. Implemente un sistema de cache utilizando Map para almacenar resultados de funciones
2. Debe incluir las siguientes funciones:
   - `cacheFunction(fn)`: Recibe una función y devuelve una nueva función que cachea resultados
   - `getCache()`: Devuelve el Map con todos los resultados cacheados
   - `clearCache()`: Limpia el cache
3. Aplica este sistema a una función que calcule el factorial de un número

Crea funciones adicionales para:

- Medir y comparar los tiempos de ejecución
- Configurar el tamaño máximo de la cache
- Aplicar el sistema a otras funciones complejas

Muestra los resultados por consola.

Ejercicio 10: Gestión de Permisos con Set

Objetivo: Practicar el uso de Set para gestionar permisos de usuarios.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa una función llamada `gestionarPermisos` que:

1. Cree un Map donde la clave sea el id de usuario y el valor sea un Set con sus permisos
2. Inicialice el Map con permisos básicos para los usuarios del array `usuarios`:
   - Usuarios activos: permisos de "leer", "editar"
   - Usuarios inactivos: solo permiso de "leer"
3. Implemente las siguientes funciones:
   - `agregarPermiso(idUsuario, permiso)`: Añade un permiso a un usuario
   - `eliminarPermiso(idUsuario, permiso)`: Elimina un permiso de un usuario
   - `tienePermiso(idUsuario, permiso)`: Comprueba si un usuario tiene un permiso

- `usuariosConPermiso(permiso)`: Devuelve un array con ids de usuarios que tienen un permiso

4. Devuelva un objeto con el Map de permisos y las funciones para gestionarlo

Crea funciones adicionales para:

- Crear roles predefinidos de permisos
- Asignar roles a múltiples usuarios a la vez
- Generar un informe de permisos

Muestra los resultados por consola.

Ejercicios de LocalStorage

Ejercicio 11: Persistencia de Datos de Usuario

Objetivo: Practicar el uso de LocalStorage para persistir datos.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `persistenciaUsuario` que:

1. Implemente las siguientes funciones:
   - `guardarUsuario(usuario)`: Guarda un usuario en LocalStorage
   - `obtenerUsuario(id)`: Recupera un usuario de LocalStorage por su id
   - `actualizarUsuario(id, datos)`: Actualiza datos de un usuario en LocalStorage
   - `eliminarUsuario(id)`: Elimina un usuario de LocalStorage
   - `listarUsuarios()`: Devuelve todos los usuarios guardados en LocalStorage
2. Maneja los casos en que no haya datos en LocalStorage
3. Incluye manejo de errores para operaciones que puedan fallar

Crea funciones adicionales para:

- Validar los datos antes de guardar
- Exportar/importar datos en formato JSON
- Sincronizar datos entre diferentes pestañas

Muestra los resultados por consola.

Ejercicio 12: Historial de Búsquedas

Objetivo: Practicar el uso de LocalStorage para mantener historial.

Datos necesarios: Array `palabras` del archivo `src/db/data.js` para demostración

Enunciado: Implementa un módulo llamado `historialBusquedas` que:

1. Implemente un sistema de historial de búsquedas utilizando LocalStorage
2. Debe incluir las siguientes funciones:
    - `agregarBusqueda(termino)`: Añade una nueva búsqueda al historial
    - `obtenerHistorial()`: Devuelve el historial completo ordenado por fecha (más reciente primero)
    - `eliminarBusqueda(id)`: Elimina una búsqueda específica del historial
    - `limpiarHistorial()`: Elimina todo el historial
    - `buscarEnHistorial(termino)`: Busca búsquedas que contengan un término
3. Cada búsqueda debe guardar: término, fecha y un id único
4. Limita el historial a un máximo de 50 búsquedas

Crea funciones adicionales para:

- Agrupar búsquedas por términos similares
- Marcar búsquedas como favoritas
- Calcular estadísticas de las búsquedas más frecuentes

Muestra los resultados por consola.

Ejercicio 13: Configuración de Preferencias

Objetivo: Practicar el uso de LocalStorage para guardar configuraciones.

Datos necesarios: No se requieren datos específicos del archivo data.js

Enunciado: Implementa un módulo llamado `configuracionPreferencias` que:

1. Implemente un sistema de preferencias de usuario utilizando LocalStorage
2. Debe incluir las siguientes funciones:
    - `establecerPreferencia(clave, valor)`: Guarda una preferencia
    - `obtenerPreferencia(clave, valorPorDefecto)`: Obtiene una preferencia con valor por defecto
    - `restablecerPreferencias()`: Restablece todas las preferencias a sus valores por defecto
    - `exportarPreferencias()`: Devuelve un objeto con todas las preferencias
    - `importarPreferencias(preferencias)`: Importa un objeto de preferencias

3. Define preferencias por defecto para:
    - Tema (claro/oscuro)
    - Idioma (es/en/fr)
    - Notificaciones (activadas/desactivadas)
    - Elementos por página (10/25/50)

Crea funciones adicionales para:

- Validar los valores antes de guardar
- Crear perfiles de configuración
- Aplicar las preferencias automáticamente

Muestra los resultados por consola.

Ejercicio 14: Carrito de Compras Persistente

Objetivo: Practicar el uso de LocalStorage para un carrito de compras.

Datos necesarios: Array `productos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `carritoCompras` que:

1. Implemente un carrito de compras que persiste en LocalStorage
2. Debe incluir las siguientes funciones:
    - `agregarProducto(idProducto, cantidad)`: Añade un producto al carrito
    - `eliminarProducto(idProducto)`: Elimina un producto del carrito
    - `actualizarCantidad(idProducto, nuevaCantidad)`: Actualiza la cantidad de un producto
    - `vaciarCarrito()`: Elimina todos los productos del carrito
    - `obtenerCarrito()`: Devuelve el contenido actual del carrito
    - `calcularTotal()`: Calcula el precio total del carrito
3. Utiliza los datos del array `productos` para obtener información de los productos
4. Guarda en el carrito: idProducto, nombre, precio, cantidad y subtotal

Crea funciones adicionales para:

- Aplicar códigos de descuento
- Calcular gastos de envío según el total
- Simular el proceso de checkout

Muestra los resultados por consola.

Ejercicio 15: Sistema de Favoritos

Objetivo: Practicar el uso de LocalStorage para gestionar elementos favoritos.

Datos necesarios: Arrays `usuarios` y `productos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `sistemaFavoritos` que:

1. Implemente un sistema de favoritos para usuarios y productos utilizando LocalStorage
2. Debe incluir las siguientes funciones:
   - `agregarFavorito(tipo, id)`: Añade un elemento a favoritos (tipo: 'usuario' o 'producto')
   - `eliminarFavorito(tipo, id)`: Elimina un elemento de favoritos
   - `esFavorito(tipo, id)`: Comprueba si un elemento está en favoritos
   - `obtenerFavoritos(tipo)`: Devuelve todos los favoritos de un tipo
   - `obtenerFavoritosConDetalles(tipo)`: Devuelve los favoritos con todos sus detalles
3. Utiliza los arrays `usuarios` y `productos` para obtener los detalles
4. Cada favorito debe guardar: tipo, id y fecha de adición

Crea funciones adicionales para:

- Filtrar favoritos por fecha de adición
- Buscar dentro de los favoritos
- Crear colecciones personalizadas de favoritos

Muestra los resultados por consola.

Ejercicios de Funciones

Ejercicio 16: Generador de Datos de Productos

Objetivo: Practicar la creación de funciones para procesar datos de productos.

Datos necesarios: Array `productos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `procesadorProductos` que:

1. Implemente las siguientes funciones:
   - `generarResumenProducto(producto)`: Crea un resumen de un producto específico
   - `generarResumenProductos(listaProductos)`: Genera resúmenes para múltiples productos

- `aplicarFiltros(productos, filtros)`: Filtra productos según criterios
- `ordenarProductos(productos, criterio)`: Ordena productos según diferentes criterios

2. Cada resumen debe incluir: nombre, categoría, precio y stock
3. Añade etiquetas diferentes según el stock (alto, medio, bajo)
4. Crea filtros para: categoría, rango de precios y disponibilidad
5. Demuestra su funcionamiento con diferentes productos

Muestra los resultados por consola.

Ejercicio 17: Sistema de Tablas de Datos

Objetivo: Practicar la creación de tablas de datos y funciones de ordenación.

Datos necesarios: Array `usuarios` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `tablaDatos` que:

1. Implemente las siguientes funciones:
   - `crearTabla(datos, columnas)`: Crea una estructura de tabla a partir de datos
   - `ordenarTabla(datos, columna, direccion)`: Ordena los datos según una columna
   - `paginarTabla(datos, pagina, elementosPorPagina)`: Implementa paginación
   - `filtrarTabla(datos, terminoBusqueda)`: Filtra datos por un término
   - `exportarTabla(datos, formato)`: Exporta los datos en diferentes formatos
2. La tabla debe ser procesable con:
   - Ordenación por diferentes columnas
   - Controles de paginación
   - Búsqueda de texto
   - Resaltado de datos específicos
3. Demuestra su funcionamiento con los datos de usuarios

Muestra los resultados por consola.

Ejercicio 18: Gestor de Validación de Datos

Objetivo: Practicar la creación de funciones de validación de datos.

Datos necesarios: Arrays `usuarios` y `productos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `validadorDatos` que:

1. Implemente las siguientes funciones:
   - `validarCampo(tipo, valor)`: Valida diferentes tipos de datos
   - `validarUsuario(usuario)`: Valida un objeto de usuario completo
   - `validarProducto(producto)`: Valida un objeto de producto completo
   - `mostrarErrores(errores)`: Formatea los errores de validación
   - `corregirDatos(datos, tipo)`: Intenta corregir datos comunes
2. Crea validaciones para:
   - Email (formato válido)
   - Edad (rango específico)
   - Precio (número positivo)
   - Campos requeridos
3. Demuestra su funcionamiento validando diferentes datos

Muestra los resultados por consola.

Ejercicio 19: Sistema de Notificaciones por Consola

Objetivo: Practicar la creación de un sistema de notificaciones por consola.

Datos necesarios: No se requieren datos específicos del archivo data.js

Enunciado: Implementa un módulo llamado `sistemaNotificaciones` que:

1. Implemente las siguientes funciones:
   - `mostrarNotificacion(mensaje, tipo)`: Muestra una notificación por consola
   - `formatearNotificacion(mensaje, tipo)`: Formatea una notificación según su tipo
   - `registrarNotificacion(mensaje, tipo)`: Registra una notificación en un historial
   - `obtenerHistorialNotificaciones()`: Devuelve el historial de notificaciones
   - `filtrarNotificaciones(tipo)`: Filtra notificaciones por tipo
2. Tipos de notificaciones:
   - éxito (verde)
   - error (rojo)
   - advertencia (amarillo)
   - información (azul)
3. Características:
   - Formato especial para cada tipo

- Incluir timestamp en cada notificación
- Historial de notificaciones

4. Demuestra su funcionamiento mostrando diferentes tipos de notificaciones

Muestra los resultados por consola.

Ejercicio 20: Dashboard de Datos

Objetivo: Practicar la creación de un dashboard con visualización de datos por consola.

Datos necesarios: Arrays `usuarios`, `productos` y `pedidos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `dashboardDatos` que:

1. Implemente las siguientes funciones:
   - `crearResumenUsuarios()`: Crea un resumen de estadísticas de usuarios
   - `crearResumenProductos()`: Crea un resumen de estadísticas de productos
   - `crearResumenPedidos()`: Crea un resumen de estadísticas de pedidos
   - `actualizarDashboard()`: Actualiza todos los datos del dashboard
   - `exportarDatos(formato)`: Exporta los datos del dashboard
2. El dashboard debe incluir:
   - Estadísticas de usuarios por ciudad
   - Estadísticas de productos por categoría
   - Estadísticas de pedidos por estado
   - Resumen de métricas clave
3. Características adicionales:
   - Formato tabular para los datos
   - Opciones de filtrado
   - Exportación a formato legible
4. Demuestra su funcionamiento mostrando diferentes visualizaciones de datos

Muestra los resultados por consola.

Ejercicio 21: Sistema de Recomendaciones Inteligente

Objetivo: Practicar el uso combinado de Map, Set, arrays de objetos y funciones complejas para crear un sistema de recomendaciones basado en comportamiento de usuarios.

Datos necesarios: Arrays `usuarios`, `productos` y `pedidos` del archivo `src/db/data.js`

Enunciado: Implementa un módulo llamado `sistemaRecomendaciones` que:

1. Analice el historial de compras y preferencias de los usuarios para generar recomendaciones personalizadas
2. Implemente las siguientes funciones:
   - `analizarPatronesCompra()`: Crea un Map que relacione cada usuario con las categorías que ha comprado y su frecuencia
   - `calcularSimilitudUsuarios(idUsuario1, idUsuario2)`: Calcula un índice de similitud entre dos usuarios basado en:
     - Hobbies compartidos (peso: 30%)
     - Categorías de productos comprados en común (peso: 40%)
     - Diferencia de edad (peso: 15%)
     - Ciudad (peso: 15%)
   - `obtenerUsuariosSimilares(idUsuario, limite)`: Devuelve los N usuarios más similares al usuario dado
   - `generarRecomendaciones(idUsuario, numeroRecomendaciones)`: Genera recomendaciones de productos basadas en:
     - Productos comprados por usuarios similares que el usuario objetivo no ha comprado
     - Productos de categorías que el usuario ha comprado previamente con alta valoración
     - Productos destacados que coincidan con los hobbies del usuario
   - `obtenerEstadisticasRecomendaciones()`: Devuelve estadísticas del sistema (usuarios analizados, productos recomendables, etc.)
3. El sistema debe considerar:
   - Solo recomendar productos con stock disponible
   - Priorizar productos con valoración >= 4.0
   - Evitar recomendar productos ya comprados por el usuario
   - Guardar en caché las similitudes calculadas usando Map para optimizar rendimiento
4. Crea funciones adicionales para:
   - `actualizarRecomendaciones(idUsuario)`: Actualiza las recomendaciones cuando hay nuevos datos
   - `explicarRecomendacion(idUsuario, idProducto)`: Explica por qué se recomienda un producto específico
   - `obtenerTendencias()`: Identifica productos y categorías más populares entre usuarios similares
   - `guardarRecomendacionesLocalStorage(idUsuario)`: Persiste las recomendaciones generadas

- `compararEfectividad()`: Simula qué porcentaje de recomendaciones podrían ser aceptadas

5. El resultado de `generarRecomendaciones` debe devolver un array de objetos con esta estructura:

```
{
  producto: {...}, // objeto producto completo
  puntuacion: número, // de 0 a 100
  razones: ["razón1", "razón2", ...], // por qué se recomienda
  usuariosSimilaresQueCompraron: [ids], // usuarios similares que lo compraron
  categoriaRelacionada: string
}
```

Demuestra el funcionamiento:

- Genera recomendaciones para al menos 3 usuarios diferentes
- Muestra las similitudes entre usuarios
- Explica por qué se recomienda cada producto
- Muestra estadísticas del sistema
- Compara diferentes configuraciones de ponderación

Muestra los resultados por consola usando `console.group()`, `console.table()` y formato visual claro.

Instrucciones Generales

1. **Importación de datos:** Para resolver los ejercicios, importa los datos necesarios desde `src/db/data.js`:

```
import {
  usuarios,
  productos,
  pedidos,
  numeros,
  ciudadesPoblacion,
  coloresPrimarios,
  coloresSecundarios,
} from "./src/db/data.js";
```

   Cada ejercicio especifica qué datos necesitas utilizar.

2. **Estructura de archivos:** Crea un archivo por cada ejercicio o agrupa varios ejercicios relacionados en un mismo archivo.

3. **Testing:** Para cada ejercicio, incluye ejemplos de uso que demuestren que las funciones funcionan correctamente.

4. **Buenas prácticas:** Aplica las buenas prácticas de JavaScript modernas:

    - Usa arrow functions cuando sea apropiado
    - Desestructura objetos y arrays para facilitar el trabajo
    - Usa métodos funcionales de arrays (map, filter, reduce, etc.)
    - Maneja casos extremos y errores con try/catch
    - Documenta las funciones con comentarios claros
    - Usa sintaxis ES6+ (template literals, destructuring, spread operator, etc.)

5. **Salida de datos:** Todos los ejercicios deben mostrar los resultados por consola:

    - Usa `console.log()` para mostrar resultados
    - Usa `console.table()` para mostrar arrays y objetos de forma tabular
    - Usa `console.group()` y `console.groupEnd()` para agrupar mensajes relacionados
    - Incluye mensajes descriptivos que expliquen lo que se está mostrando

6. **Desafío adicional:** Una vez completados todos los ejercicios, intenta crear una pequeña aplicación que integre varias de las funcionalidades implementadas.