Documentación - Informe técnico del desarrollo del Trabajo Práctico grupal de e-commerce

Instituto: ISTEA

Materia: Laboratorio de Aplicaciones Web Cliente

Docente: Jesús, Carlos

Integrantes:

- Andrés Valdés
- Eric Escobar
- Santiago Andini
- Francisco Acuña

Descripción: El presente documento es el producto de la unión de los informes de cada uno de los integrantes explicando su participación en el proyecto.

Andrés Valdés – Contribución al proyecto de e-commerce

1. Introducción

Este documento detalla mi contribución al proyecto de e-commerce para la materia “Laboratorio de Aplicaciones Web”. Me encargué del desarrollo de la estructura base del sitio, incluyendo la creación del archivo index.html, la maquetación inicial con Bootstrap, la implementación del buscador, el enrutamiento entre secciones y la funcionalidad de filtrado por categorías utilizando JavaScript.

2. Estructura de archivos

Mi trabajo se dividió principalmente entre los siguientes archivos:
- index.html: Maquetación principal de la home, navegación, buscador y pie de página.
- css/style.css: Estilos personalizados para mejorar la presentación del buscador y layout general.
- js/main.js: Lógica para el filtrado dinámico por categorías.

3. index.html: Estructura principal

3.1. Funcionalidad general

• Incluye el encabezado con la barra de navegación y botones para filtrar productos por categoría.
• Sección con input de búsqueda.
• Sección de productos que se completa dinámicamente.
• Modal para producto individual (estructura base).
• Sidebar de carrito (offcanvas) con botones para vaciar y finalizar compra.
• Pie de página con enlaces.

3.2. Botones de categoría

Cada botón tiene una clase 'categoria' y un ID correspondiente al nombre de la categoría en la base de datos:

```
<button class="btn categoria" id="men's clothing">Ropa de Hombre</button>
<button class="btn categoria" id="jewelery">Joyeria</button>
<button class="btn categoria" id="electronics">Electronica</button>
<button class="btn categoria" id="women's clothing">Ropa de Mujer</button>
```

4. main.js: Filtrado de categorías

4.1. Funcionalidad general

• Escucha los clics en los botones de categoría y filtra los productos en pantalla.
• Si se selecciona "all", muestra todos los productos.

4.2. Código desarrollado

```
const botones = document.querySelectorAll('.categoria');
botones.forEach(boton => {
  boton.addEventListener('click', () => {
    if (boton.id !== 'all') {
```

```
    const categoria = boton.id;
    let filterProducts = allProducts.filter(p => p.category === categoria);
    rederizarPaginaConProductos(filterProducts);
   } else {
    let filterProducts = allProducts;
    rederizarPaginaConProductos(filterProducts);
   }
  });
});
```

4.3. Integración

Este script está vinculado directamente con los botones del index.html, lo que permite a los usuarios visualizar productos por categoría de forma dinámica.

5. style.css: Estilización personalizada

• Se personalizó el input del buscador (.custom-search) para reducir su ancho, centrarlo en la página y agregarle bordes más visibles:

```
.search-wrapper {
  display: flex;
  justify-content: center;
}

.custom-search {
  max-width: 400px;
  border: 2px solid #ccc;
  border-radius: 8px;
}
```

6. Conclusión

Mi contribución permitió establecer la base visual y funcional del e-commerce, asegurando que el sitio sea navegable, visualmente agradable y funcional mediante el filtrado de productos. El uso de Bootstrap permitió acelerar la maquetación, y el uso de JavaScript hizo posible la interacción dinámica sin recargar la página.

Eric Escobar – Buscador

Esta función hace que el buscador funcione en tiempo real. Cuando el usuario empieza a escribir, se va filtrando la lista de productos, solo se muestra los productos que coinciden y si el usuario borra el texto, se muestra todos los productos nuevamente.

```
let allProducts = [];

function setupSearchBar() {
```

-Esta función se encarga de activar el buscador mientras el usuario escribe en el campo de búsqueda.

```
const searchInput = document.querySelector('#searchInput');
```

-En esta parte lo que hago es guardar en una constante el input del HTML donde el usuario va a escribir lo que quiere buscar, que lo selecciona por su ID que es searchInput.

```
searchInput.addEventListener('input', (event) => {
```

-Después agrego un evento input, que se activa cada vez que el usuario escribe algo en el campo. Este evento se va ejecutar en una función cada vez que cambia el texto

```
const searchText = event.target.value.toLowerCase();
```

- Dentro del evento agarro el texto que el usuario escribió, con event.target.value lo paso a minúsculas para que no exista problemas si se usó mayúsculas o minúsculas y lo guardo en una constante llamada searchText.

```
if (searchText === "") {
    rederizarPaginaConProductos(allProducts);
  }
```

- Si el campo está vacío entonces llamo a la función rederizarPaginaConProductos pasando todos los productos (allProducts). Esto sirve para volver a mostrar todo sin filtros.

```
else {
    const filteredProducts = allProducts.filter(product =>
```

```
        product.title.toLowerCase().includes(searchText)
      );
```

- Pero si se escribió algo, entonces uso .filter para buscar dentro del array de productos (allProducts). Con lo que comparo el título de cada producto con el texto que escribió el usuario para ver si el nombre del producto está.

```
      rederizarPaginaConProductos(filteredProducts);
   }
 });
}
```

- En la parte final se actualiza la página mostrando solo los productos que coinciden con la búsqueda.

function setupSearchButton() {

Esta función sirve para que el botón de búsqueda funcione cuando el usuario hace clic en él.

```
const searchInput = document.getElementById('searchInput');
const searchButton = document.getElementById('btnBuscar');
```

- En esta parte selecciono dos elementos del HTML que serían el campo de texto donde el usuario escribe lo que quiere buscar (searchInput) y el botón que tiene el ícono de la lupa (btnBuscar)

```
searchButton.addEventListener('click', () => {
```

- Después le agrego un evento al botón y este evento se ejecuta cuando el usuario hace clic en él.

```
const searchText = searchInput.value.toLowerCase();
```

- Dentro del evento tomo el valor que el usuario escribió en el input, lo paso a minúsculas y lo guardo en una variable llamada searchText.

```
if (searchText === "") {
```

```
    rederizarPaginaConProductos(allProducts);
  }
```

- Si el input está vacío, eso significa que no escribió nada, por lo cual muestro los productos usando rederizarPaginaConProductos(allProducts).

```
else {
    const filteredProducts = allProducts.filter(product =>
      product.title.toLowerCase().includes(searchText)
    );
```

- Pero si escribió algo, filtro los productos con .filter() y me fijo si el title de cada producto incluye el texto que el usuario puso. Si coincide lo guardo en un nuevo array llamado filteredProducts.

```
rederizarPaginaConProductos(filteredProducts);
  }
 });
}
```

- Por último, se actualiza la página, mostrando solo los productos filtrados.

Santiago Andini – Renderizado de productos

Yo hice el renderizado de la página con los productos.

Cuando el usuario entra a la página web se recuperan los productos por la API. Eso se asigna a una variable global que se pasa a la function renderizar productos. Ahí, si la variable tiene contenido (el length no devuelve falso), se toma la etiqueta de HTML que tiene el id productos con el método querySelector. En el caso de que tenga algo asignado se la limpia con el primer innerHTML y luego mediante un bucle se crea un elemento div al que se le agregan una serie de clases. Luego, dentro del innerHTML de ese elemento se agregan las cards de cada producto. Al finalizar se agrega a la seccion con el método appendChild y se configura el botón de mostrar detalles tomando el id dinámico creado en la creación de las cards y se le asigna con el método onClick la function mostrarDetalles. En el caso de que el length de los productos sea falso, se carga un mensaje que avisa que no hay productos.

La función mostrarDetalles toma el div con el id myModal y le asigna en su innerHTML el estilo y la información sobre el producto de forma más detallada también permite agregar al producto al carrito.

Estas dos funciones llaman a la función updateCartCounter para que el botón de la vista previa del carrito se actualice con la cantidad de productos comprados (no con el total).

Francisco Acuña – Contribución al proyecto de e-commerce

---

1. Introducción

Este documento describe con detalle mi contribución al proyecto de e-commerce de la materia "Laboratorio de Aplicaciones Web". Mi responsabilidad fue la **implementación de la lógica del carrito** empleando localStorage (módulo **js/cartService.js**) y la **gestión de su interfaz** (módulo **js/cartUI.js**). A continuación, se detalla cada componente, su funcionamiento y cómo se integran.

---

2. Estructura de archivos

En la carpeta js/ entregué dos archivos principales de mi parte:

- **cartService.js**
    - Lógica de negocio: manejo de localStorage, contador y eventos.
- **cartUI.js**
    - Presentación: escucha eventos y renderiza el sidebar del carrito.

---

3. cartService.js: Lógica de negocio del carrito

3.1. Funcionalidad general

- Gestiona el carrito en localStorage bajo la clave "cart".
- Proporciona funciones para:
    - Agregar y actualizar productos.
    - Cambiar cantidades y eliminar ítems.
    - Vaciar el carrito.
    - Actualizar el contador visible en la barra.
- Emite un evento cartUpdated tras cada cambio para sincronizar la UI.

3.2. Funciones exportadas

1. **getCart(): Array**
    - Lee y parsea localStorage.getItem('cart').
    - Retorna un array de objetos { id, title, price, image, quantity } o [] si no existe.
2. **updateCartCounter(): void**
    - Llama a getCart() y actualiza el contenido del badge #cart-count con la longitud del carrito.
3. **saveCart(cart: Array): void**
    - Convierte el array cart a JSON y lo guarda en localStorage.
4. **addToCart(product: Object): void**
    - Recibe un objeto mínimo { id, title, price, image }.
    - Incrementa quantity si el producto existe, o lo añade con quantity = 1.
    - Muestra una notificación tipo toast (SweetAlert2) cuando se añade por primera vez.
    - Llama a saveCart(), updateCartCounter() y dispatchCartUpdated().
5. **changeQuantity(id: number|string, delta: number): void**
    - Modifica la cantidad (quantity) del producto con el id dado sumando delta.
    - Asegura que quantity >= 1.
    - Llama a saveCart(), dispatchCartUpdated().
6. **removeFromCart(id: number|string): void**
    - Elimina el producto completo cuyo id coincide.
    - Llama a saveCart(), dispatchCartUpdated() y updateCartCounter().
7. **clearCart(): void**
    - Vacía el carrito (reemplaza con []).
    - Llama a saveCart(), dispatchCartUpdated() y updateCartCounter().
8. **dispatchCartUpdated(cart: Array): void** (privada)
    - Crea y despacha new CustomEvent('cartUpdated', { detail: cart }).

---

4. cartUI.js: Presentación y eventos

4.1. Funcionalidad general

- Está vinculado al contenedor #cart-container y a los botones:
    - #cart-clear-btn (vaciar carrito)
    - #cart-checkout-btn (finalizar compra).
- Inicializa la vista al cargar (DOMContentLoaded) y con cada evento cartUpdated.
- Renderiza cada ítem del carrito con:
    - Imagen, título y precio total (price * quantity).
    - Botones **–**, **+** y **Eliminar** con sus handlers.
- Controla la habilitación de botones de vaciar y finalizar según haya o no ítems.
- En el botón de finalizar compra muestra una alerta (SweetAlert2).

4.2. Función renderCart(cart: Array)

1. Limpia el contenedor (innerHTML = '').
2. Por cada item en cart:
    - Crea un div con estructura HTML y tres botones.
    - Obtiene botones con querySelectorAll('button').
    - Asigna:
      btnMinus.onclick = () => changeQuantity(item.id, -1);
      btnPlus.onclick = () => changeQuantity(item.id, +1);
      btnDel.onclick = () => removeFromCart(item.id);
    - Inserta el nodo en #cart-container.
3. Activa/desactiva clearBtn y checkoutBtn según cart.length.

4.3. Botones adicionales

- **Vaciar carrito** → clearCart().
- **Finalizar compra** → clearCart() + alerta de éxito (SweetAlert2).

---

5. Conclusión

Mi parte implementa **toda la persistencia**, **conteo** y **síncronia** del carrito mediante localStorage, eventos y un contador de carrito visible. También provee una interfaz reactiva para que el usuario agregue, modifique y elimine productos con feedback inmediato.
