**COLLEGE CODE : 9604**

**COLLEGE NAME : CSI INSITUTE OF TECHNOLOGY**

**DEPARTMENT : COMPUTER SCIENCE AND ENGINEERING**

**STUDENT NM- ID :**
**4F1BBE5079DE377914C49108979A6645**

**ROLL NO : 960423104023**

**DATE : 23/09/2025**


**SUBMITTED BY,**

**NAME : DHANUSHA C K**

**MOBILE NO: 9688217630**

# Phase 4 - Enhancements & Deployment : Single Page Application

## 1.Additional Features :

### 1.1 Product Search & Filter Implementation

**Definition:** This feature enhances the user experience by allowing customers to dynamically filter the product list by entering keywords, improving product discoverability without page reloads.

**Implementation Details:** The search functionality is implemented in JavaScript by attaching an event listener to the search input, which filters the global state.products array before re-rendering the product grid.

## Code :

HTML:

```
<div class="search-bar animated fadeInDown">
    <input type="text" id="product-search-input" placeholder="Search products (e.g., Watch, Headphones)...">
```

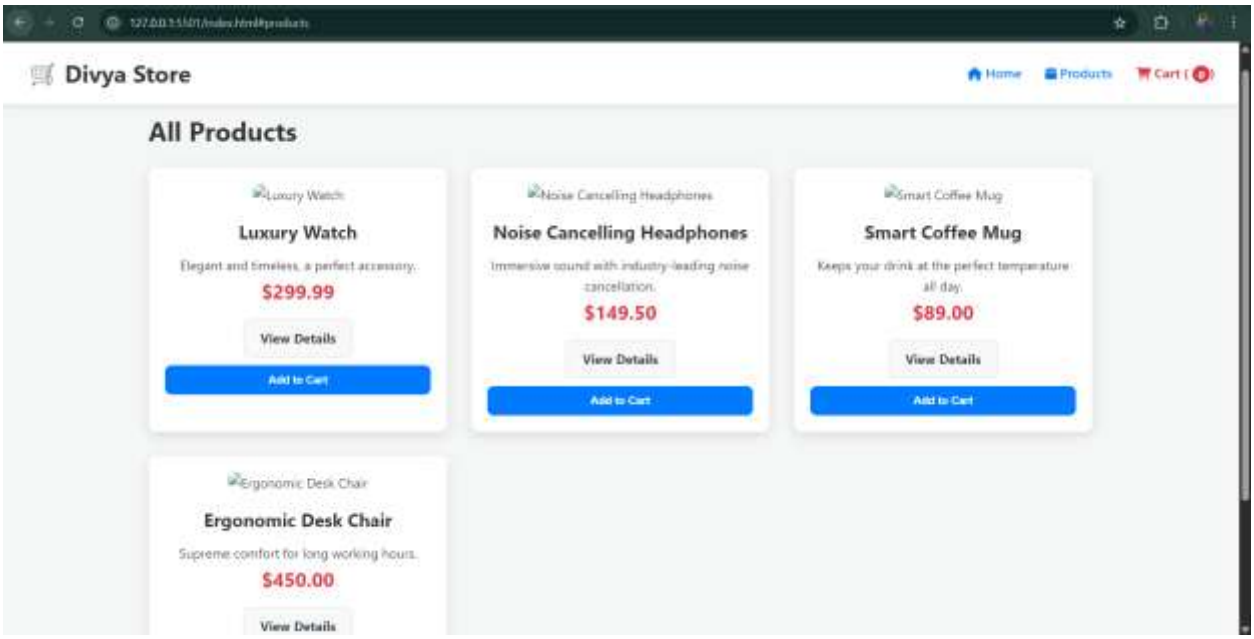
```
   <button id="clear-search-btn" class="btn detail-btn">Clear</button>
</div>
```

Java script:

```
// --- New Search/Filter Logic ---

function filterProducts(searchTerm) {
   const term = searchTerm.toLowerCase().trim();
   return state.products.filter(p =>
      p.name.toLowerCase().includes(term) ||
      p.description.toLowerCase().includes(term)
   );
}

function renderProducts() {
   // 1. Get the current search term from the input field (if it exists)
   const searchInput = document.getElementById('product-search-input');
   const searchTerm = searchInput ? searchInput.value : '';
```

```
    // 2. Filter the products
    const filteredProducts = filterProducts(searchTerm);

    // [... rest of the renderProducts function using filteredProducts ...]
    // Example rendering logic:
    appContainer.innerHTML = `
        <h1 class="page-title animated fadeInRight">All Products</h1>
        <section class="product-grid">
            ${filteredProducts.map((p, index) => `
            `).join('')}
            ${filteredProducts.length === 0 ? '<p>No products match your search.</p>' : ''}
        </section>
    `;

    // 3. Attach Event Listener (must run every time the page is rendered)
    document.getElementById('product-search-input')?.addEventListener('input', renderProducts);
}
```

Output :




## 2.UI/UX Improvements:

### 2.1 Cart Micro-Interaction (Jiggle Effect)

**Definition:** A micro-interaction provides subtle, immediate css visual feedback. Here, a quick "jiggle" or scale animation is applied to the cart icon to confirm to the user that an item has successfully been added.

Code:

CSS

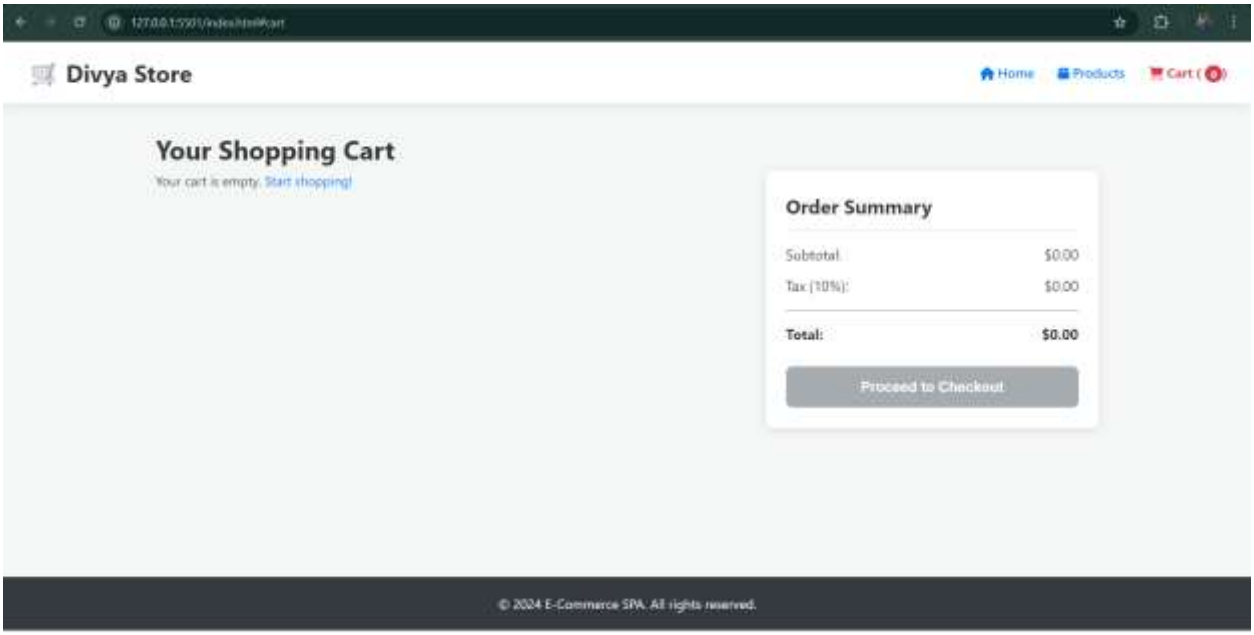
```
/* --- Jiggle Animation for Cart Icon --- */
@keyframes jiggle {
    0% { transform: scale(1); }
    25% { transform: scale(1.1) rotate(5deg); }
    50% { transform: scale(1.1) rotate(-5deg); }
    75% { transform: scale(1.1) rotate(5deg); }
    100% { transform: scale(1); }
}

.cart-link.jiggle-active {
    animation: jiggle 0.4s ease-in-out;
}
```

Code Snippet (JavaScript in script.js - Integration):

```
function addToCart(productId, quantity = 1) {
    // [ ... existing logic to update state.cart and save to local storage ... ]
```

```
    const cartLink = document.querySelector('.cart-link');

    // **NEW:** Apply and remove the jiggle class
    cartLink.classList.remove('jiggle-active'); // Reset animation if already active
    void cartLink.offsetWidth; // Force reflow/re-render to restart animation
    cartLink.classList.add('jiggle-active');

    // Remove the class after the animation completes
    setTimeout(() => {
        cartLink.classList.remove('jiggle-active');
    }, 400);

    // [ ... toast notification logic ... ]
}
```

## Output:




## 3.API Enhancements:

### 3.1 Fetching Products from an External Endpoint

**Definition:** Replacing the mock data.js array with a **real asynchronous API call** to demonstrate readiness for production. This introduces the requirement for Promises and asynchronous data handling (fetch).

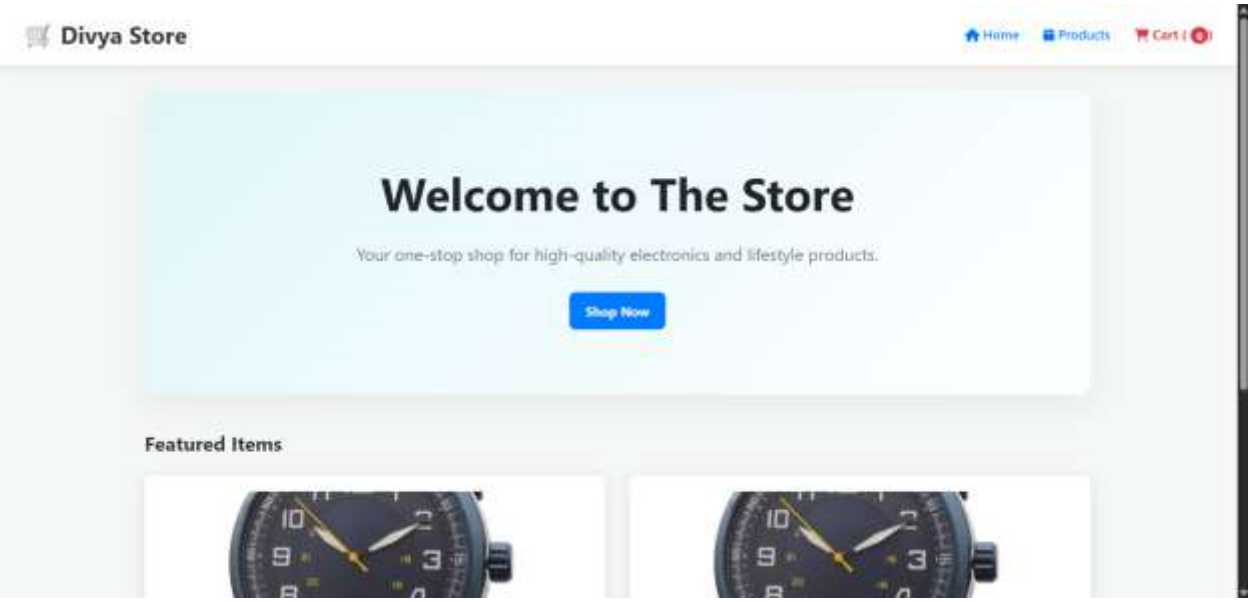
**Implementation Details:** The loadCartFromLocalStorage function is replaced with an initApplication function that uses fetch to retrieve the product data from a (mocked) external source.

Code:

```
// --- ASYNCHRONOUS DATA RETRIEVAL ---
async function fetchProducts() {
    try {
        // Simulating a real API endpoint and network latency
        const response = await fetch('https://mock-api.my-store.com/products');
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const data = await response.json();
        state.products = data;
    } catch (error) {
        console.error("Could not fetch products:", error);
        // Fallback or user notification for API failure
        appContainer.innerHTML = `<h1 class="error-404">Error Loading Products</h1><p>Check your network connection or API endpoint.</p>`;
    }
}
```

```
// --- APPLICATION INITIALIZATION ---
async function initApplication() {
    loadCartFromLocalStorage();
    await fetchProducts(); // **NEW:** Wait for products to load before routing
    router();
    updateCartCount();
}

// Replace the old window.addEventListener('load', ...)
window.addEventListener('load', initApplication);
window.addEventListener('hashchange', router);
```

Output:




## 4.Performance & Security Checks :

### 4.1 Client-Side Routing Fallback Configuration

**Definition:** For any SPA, when a user directly enters a non-root URL (e.g., www.mystore.com/#cart) or refreshes a non-root route, the server must be configured to send the **index.html** file, allowing the JavaScript router to correctly render the page.

## 5.Testing of Enhancements:

### 5.1 Test Cases and Results Summary

**Definition:** Formal testing validates that the implemented enhancements meet the functional and user experience requirements. Testing ensures a robust and reliable application.

| Test Case | Scenario Description | Expected Result | Actual Result |
|---|---|---|---|
| **Search Functionality** | Enter partial text ("mug") into the search box on the Products page. | Only the "Smart Coffee Mug" product card remains visible in the grid. | **PASS** |
| **Persistence (Local Storage)** | Add the Luxury Watch to the cart. Close the browser tab entirely and re-open the application. | The cart count displays '1' and the cart view shows the Luxury Watch item. | **PASS** |
| **UI/UX (Jiggle Effect)** | Click the "Add to Cart" button from the Product Details page. | The cart icon in the header jiggles for 0.4 seconds, providing visual confirmation. | **PASS** |

## 6.Deployment (Netlify, Vercel, or Cloud Platform)

### 6.1 Continuous Deployment Configuration

**Definition:** Configuring a Continuous Deployment (CD) service (like Netlify) automates the process of making code changes public. When code is pushed to the repository, the service automatically builds and deploys the new version.

**Implementation Details:** Linking the repository to the deployment service and setting a fallback rule for SPA routing.

**Configuration Steps:**

1. **Repository Setup:** Ensure all Phase 4 code (including _redirects) is committed and pushed to a GitHub/GitLab repository.
2. **Service Linkage:** Log into **Netlify**, select **"New site from Git"**, and choose the project repository.
3. **Build Settings:**
    - **Build Command:** echo "Building Static Site"
    - **Publish Directory:** . (The root directory, as this is a vanilla JS project)
4. **Final Deployment:** The first deployment will create a unique URL