DOM & Storage

File: `js/scripts.js` *`*all code snippets*`*

```
// Dom helpers (js/scripts.js)
class Dom {
  static $(sel, root = document) { return root.querySelector(sel); }
  static $all(sel, root = document) { return
Array.from(root.querySelectorAll(sel)); }
  static make(tag, className = "", html = "") {
    const el = document.createElement(tag);
    if (className) el.className = className;
    if (html) el.innerHTML = html;
    return el;
  }
}

// Local storage gateway (js/scripts.js)
class Store {
  static getCart() {
    try { return JSON.parse(localStorage.getItem('cart')) || []; }
    catch { return []; }
  }
  static setCart(cart) { localStorage.setItem('cart',
JSON.stringify(cart)); }
  static clearCart() { localStorage.removeItem('cart'); }

  static get(key, fallback = null) {
    try { const v = localStorage.getItem(key); return v ? JSON.parse(v) :
fallback; }
    catch { return fallback; }
  }
  static set(key, value) { localStorage.setItem(key,
JSON.stringify(value)); }
  static remove(key) { localStorage.removeItem(key); }
}
```

Description:
These two classes remove repetitive code from the rest of the project. `Dom` centralizes common element operations: find one (`$`), find many (`$all`), and create (`make`). `Store` provides safe read/write helpers for `localStorage`, including the cart and any saved user data. This keeps data access consistent and easy to test.

Catalog: Load CSV

File: `js/scripts.js`

```
class Catalog {
  static allProducts = [];

  static async load(csvPath = 'data/catalog.csv') {
    const grid = Dom.$('#catalog');
    if (!grid) return;

    try {
      const res = await fetch(csvPath);
      const text = await res.text();
      const lines = text.split('\n').slice(1);

      const products = [];
      for (const line of lines) {
        if (!line.trim()) continue;
        const parts = line.split(',');
        const name = parts[0]?.trim();
        const price = parseFloat(parts[1]);
        const img = parts[2]?.trim();
        const category = parts[3]?.trim() || "Misc";
        if (!name || isNaN(price) || !img) continue;
        products.push({ name, price, img, category });
      }

      this.allProducts = products;
      this.render(products);
      this.populateCategories(products);
      this.bindToolbar();
    } catch (err) {
      console.log('Error loading products:', err);
    }
  }
```

Description:
This method downloads the product CSV, parses each line into a product record, and stores the result in `allProducts`. After loading, it renders the initial catalog and prepares the search, filter, and sort controls. Basic checks ensure each item has a name, number price, and image path.

Catalog: Render Product Cards

```
  static render(list) {
    const grid = Dom.$('#catalog');
    if (!grid) return;
    grid.innerHTML = '';
    const frag = document.createDocumentFragment();

    list.forEach(({ name, price, img }) => {
      const col = Dom.make('div', 'col-md-3 col-sm-6 mb-4');
      col.innerHTML = `
        <div class="card shadow-sm h-100">
          <img src="${img}" class="card-img-top" alt="${name}"
onerror="this.src='images/default.jpg';">
          <div class="card-body text-center">
            <h5>${name}</h5>
            <p>$${price.toFixed(2)}</p>
            <button class="btn btn-success btn-sm" data-action="add"
data-name="${name}" data-price="${price}">
              Add to Cart
            </button>
          </div>
        </div>`;
      frag.appendChild(col);
    });

    grid.appendChild(frag);

    if (!grid._boundAddHandler) {
      grid.addEventListener('click', (e) => {
        const btn = e.target.closest('button[data-action="add"]');
        if (!btn) return;
        const name = btn.getAttribute('data-name');
        const price = parseFloat(btn.getAttribute('data-price'));
        Cart.add(name, price);
        UI.updateCartCount();
      });
      grid._boundAddHandler = true;
    }
  }
```

Description:
Builds product cards and uses event delegation on the catalog grid, so one click listener can handle all current and future product buttons without rebinding. Clicking any “Add to Cart” button calls `Cart.add()` and refreshes the cart badge.

Catalog: Toolbar & Filtering

```
  static populateCategories(products) {
    const select = Dom.$('#categoryFilter');
    if (!select) return;
    const cats = [...new Set(products.map(p =>
p.category).filter(Boolean))].sort();
    cats.forEach(cat => {
      const opt = document.createElement('option');
      opt.value = cat;
      opt.textContent = cat;
      select.appendChild(opt);
    });
  }

  static bindToolbar() {
    const search = Dom.$('#searchInput');
    const catSel = Dom.$('#categoryFilter');
    const sortSel = Dom.$('#sortSelect');
    const handler = () => this.filterAndRender();
    [search, catSel, sortSel].forEach(el => el?.addEventListener('input',
handler));
  }

  static filterAndRender() {
    const search = Dom.$('#searchInput')?.value.toLowerCase() || '';
    const cat = Dom.$('#categoryFilter')?.value || '';
    const sort = Dom.$('#sortSelect')?.value || 'name-asc';

    let filtered = this.allProducts.filter(p =>
      p.name.toLowerCase().includes(search) &&
      (cat ? p.category === cat : true)
    );

    switch (sort) {
      case 'price-asc': filtered.sort((a,b)=>a.price-b.price); break;
      case 'price-desc': filtered.sort((a,b)=>b.price-a.price); break;
      case 'name-desc': filtered.sort((a,b)=>b.name.localeCompare(a.name));
break;
      default: filtered.sort((a,b)=>a.name.localeCompare(b.name)); break;
    }

    this.render(filtered);
  }
}
```

Description: `populateCategories` fills the category dropdown based on the loaded products. `bindToolbar` connects the search box, category filter, and sort menu to the same handler. `filterAndRender` applies the user's choices and reuses `render` for the updated list.

Cart: Core State and Math

```
class Cart {
  static add(name, price) {
    let cart = Store.getCart();
    let found = cart.find(i => i.name === name);
    if (found) found.quantity++;
    else cart.push({ name, price, quantity: 1 });
    Store.setCart(cart);
  }
  static changeQuantity(index, delta) {
    let cart = Store.getCart();
    if (!cart[index]) return;
    cart[index].quantity += delta;
    if (cart[index].quantity <= 0) cart.splice(index, 1);
    Store.setCart(cart);
  }
  static remove(index) {
    let cart = Store.getCart();
    cart.splice(index, 1);
    Store.setCart(cart);
  }
  static clear() { Store.clearCart(); }
  static count() { return Store.getCart().reduce((n, i) => n + i.quantity,
0); }
  static total() { return Store.getCart().reduce((s, i) => s + i.price *
i.quantity, 0); }
  static items() { return Store.getCart(); }
}
```

Description:
This is the core functionality for the cart. It supports add, quantity change, remove, and clear. The `count()` and `total()` methods compute the number of items and the sum of item totals for the UI and checkout.

Cart UI: Render Table

```
class UI {
  static updateCartCount() {
    const badge = Dom.$('#cart-count');
    if (badge) badge.textContent = Cart.count();
  }

  static renderCartTable(itemsId = 'cart-items', totalId = 'cart-total') {
    const tbody = Dom.$('#' + itemsId);
```

```
    const totalEl = Dom.$('#' + totalId);
    if (!tbody || !totalEl) return;

    const cart = Cart.items();
    tbody.innerHTML = '';
    let total = 0;

    cart.forEach((item, i) => {
      const itemTotal = item.price * item.quantity;
      total += itemTotal;

      const tr = Dom.make('tr');
      tr.innerHTML =
        '<td>' + item.name + '</td>' +
        '<td>' +
          '<button class="btn btn-sm btn-outline-secondary"
data-action="dec" data-index="' + i + '">-</button> ' +
          item.quantity +
          ' <button class="btn btn-sm btn-outline-secondary"
data-action="inc" data-index="' + i + '">+</button>' +
        '</td>' +
        '<td>$' + item.price.toFixed(2) + '</td>' +
        '<td>$' + itemTotal.toFixed(2) + '</td>' +
        '<td><button class="btn btn-danger btn-sm" data-action="remove"
data-index="' + i + '">Remove</button></td>';
      tbody.appendChild(tr);
    });
```

Description:
This first half fills the table body using the current cart items, builds a row for each item, and keeps a running total for the page. It also shows the line total for each item (price × quantity).

```
totalEl.textContent = total.toFixed(2);

    if (!tbody._boundCartHandler) {
      tbody.addEventListener('click', (e) => {
        const btn = e.target.closest('button[data-action]');
        if (!btn) return;
        const action = btn.getAttribute('data-action');
        const index = parseInt(btn.getAttribute('data-index'));
        if (Number.isNaN(index)) return;

        if (action === 'dec') Cart.changeQuantity(index, -1);
        else if (action === 'inc') Cart.changeQuantity(index, +1);
        else if (action === 'remove') Cart.remove(index);
```

```
        UI.renderCartTable(itemsId, totalId);
        UI.updateCartCount();
      });
      tbody._boundCartHandler = true;
    }
  }
}
```

Description:
The second half sets the total and binds a single click listener to the table body. That listener handles all quantity and remove buttons. After any change, it reloads the table and updates the cart badge so the display stays correct.

Coupons: Apply + Update Totals

```
const coupons = [
  { code: "SAVE10",   discount: 0.10, expires: "2025-12-31" },
  { code: "WELCOME5", discount: 0.05, expires: "2026-01-01" }
];
let activeCoupon = null;

function applyCoupon() {
  const input = document.getElementById("coupon-code");
  const message = document.getElementById("coupon-message");
  const enteredCode = input.value.trim().toUpperCase();
  const coupon = coupons.find(c => c.code === enteredCode);
  if (!coupon) { message.textContent = "❌ Invalid coupon code.";
message.className = "text-danger small text-end"; activeCoupon = null;
updateCartTotal(); return; }
  const today = new Date(), expiry = new Date(coupon.expires);
  if (today > expiry) { message.textContent = "⚠️ This coupon has
expired."; message.className = "text-warning small text-end"; activeCoupon
= null; updateCartTotal(); return; }
  activeCoupon = coupon; message.textContent = `✅ Coupon "${coupon.code}"
applied successfully!`; message.className = "text-success small text-end";
updateCartTotal();
}

function updateCartTotal() {
  const totalElement = document.getElementById("cart-total");
  let total = Cart.total();
  if (activeCoupon) total -= total * activeCoupon.discount;
  totalElement.textContent = total.toFixed(2);
}
```

Description: Validates a code, checks its expiration date, and if valid, stores the selected coupon. `updateCartTotal` recalculates the displayed total by applying the percent discount to the cart total.

Checkout & Rewards

```
function checkout() {
  let total = Cart.total();
  if (activeCoupon) total -= total * activeCoupon.discount;
  alert(`✅ Checkout complete!\nFinal total: $${total.toFixed(2)}`);

  const pointsEarned = Math.floor(total / 10);
  const user = JSON.parse(localStorage.getItem('bb_user')) || null;
  if (user) {
    user.rewards = (user.rewards || 0) + pointsEarned;
    localStorage.setItem('bb_user', JSON.stringify(user));
  }

  Cart.clear();
  UI.renderCartTable("cart-items", "cart-total");
  UI.updateCartCount();
  updateCartTotal();
}
```

Description:
Calculates the final price with any coupon applied, shows a confirmation, adds reward points for logged-in users (1 point per $10), clears the cart, and refreshes the cart display and totals.

Accounts: User Profile & Session

```
class UserAuth {
  static key = 'bb_user';
  static sessionKey = 'bb_session';
  static get() {
    try { return JSON.parse(localStorage.getItem(this.key)) || null; }
    catch { return null; }
  }
  static save(u)  { localStorage.setItem(this.key, JSON.stringify(u)); }
  static update(patch) {
    const u = this.get() || {};
    const next = { ...u, ...patch };
    this.save(next);
    return next;
  }
  static del()    { localStorage.removeItem(this.key); }
```

```
  static isLoggedIn() { return !!this.get(); }
  static login()  { try { sessionStorage.setItem(this.sessionKey, '1'); }
catch(_){} }
  static logout() { try { sessionStorage.removeItem(this.sessionKey); }
catch(_){} }
  static isSessionActive() { try { return
!!sessionStorage.getItem(this.sessionKey); } catch(_) { return false; } }
}
```

Description:
Handles the saved user profile (`bb_user`) and the current login session flag (`bb_session`). This lets the site hide personal information when logged out while keeping the saved profile available for the next login, this is only currently implemented in localstorage.

Account Form: Populate & Bind

```
class AccountPage {
  static form() { return Dom.$('#registerForm') || Dom.$('#accountForm'); }

  static populate() {
    const f = this.form(); if (!f) return;
    const u = UserAuth.get(); if (!u) return;
    const setIf = (id, v) => { const el = Dom.$(id); if (el) el.value = v
?? ''; };
    setIf('#fullName', u.fullName);
    setIf('#email',    u.email);
    setIf('#password', u.password);
    setIf('#address',  u.address);
    setIf('#billing',  u.billing);

    const h = Dom.$('#accountHeading'); if (h) h.textContent = 'Edit
Account';
    const submit = f.querySelector('[type="submit"]'); if (submit)
submit.textContent = 'Save Changes';
    Dom.$('#deleteAccount')?.classList.remove('d-none');
  }

  static bind() {
    const f = this.form(); if (!f) return;
    const delBtn = Dom.$('#deleteAccount');
    if (delBtn) delBtn.classList.toggle('d-none', !UserAuth.get());

    f.addEventListener('submit', (e) => {
      e.preventDefault();
      const val = (id) => Dom.$(id)?.value?.trim() || '';
      const payload = {
```

```
        fullName: val('#fullName'),
        email:    val('#email'),
        password: val('#password'),
        address:  val('#address'),
        billing:  val('#billing')
      };
```

Description: `populate` fills the form with saved user data and switches the UI to “Edit Account.” `bind` prepares the page, reveals the delete button when appropriate, and starts the submit handler.

Account Form: Bind cont.

```
      if (!payload.fullName || !payload.email || !payload.password) {
        alert('Name, email, and password are required.'); return;
      }
      if (UserAuth.isLoggedIn()) {
        UserAuth.update(payload); UserAuth.login(); alert('Account
updated.');
      } else {
        UserAuth.save({ id: 'u-' + Math.random().toString(36).slice(2,8),
...payload });
        UserAuth.login(); alert('Account created.'); this.populate();
      }
      NavbarUser.render();
    });

    if (delBtn) {
      delBtn.addEventListener('click', () => {
        if (!confirm('Delete your account? This cannot be undone.'))
return;
        UserAuth.del(); UserAuth.logout(); alert('Account deleted.');

['#fullName','#email','#password','#address','#billing'].forEach(sel => {
const el = Dom.$(sel); if (el) el.value = ''; });
        delBtn.classList.add('d-none');
        const h = Dom.$('#accountHeading'); if (h) h.textContent = 'Create
Your Account';
        const submit = f.querySelector('[type="submit"]'); if (submit)
submit.textContent = 'Create Account';
        NavbarUser.render();
      });
    }
  }
}
```

Description: On submit, the code checks required fields. It either updates the existing profile or creates a new one, then marks the session as logged in. The delete handler removes both the saved profile and the session, resets the form to the “Create” state, and updates the navbar.

Navbar: Status + Logout

```
class NavbarUser {
  static mount() {
    const navRight = document.querySelector('.navbar .navbar-nav.ms-auto');
    if (!navRight) return;

    if (!document.getElementById('nav-user-status')) {
      const li = document.createElement('li');
      li.id = 'nav-user-status';
      li.className = 'nav-item ms-2 d-flex align-items-center';
      li.innerHTML = '<span class="nav-link small py-0"></span>';
      navRight.insertBefore(li, navRight.firstChild);
    }
    if (!document.getElementById('nav-logout')) {
      const li = document.createElement('li');
      li.id = 'nav-logout';
      li.className = 'nav-item ms-2 d-none d-flex align-items-center';
      li.innerHTML = '<button class="btn btn-outline-light btn-sm py-0"
type="button">Logout</button>';
      const statusLi = document.getElementById('nav-user-status');
      const navR = document.querySelector('.navbar .navbar-nav.ms-auto');
      if (statusLi && navR) navR.insertBefore(li, statusLi.nextSibling);
      li.addEventListener('click', (e) => {
        const btn = e.target.closest('button'); if (!btn) return;
        UserAuth.logout(); NavbarUser.render();
      });
    }
    this.render();
  }

  static render() {
    const slot = document.querySelector('#nav-user-status .nav-link');
    const logoutLi = document.getElementById('nav-logout');
    if (!slot) return;
    const u = (typeof UserAuth !== 'undefined') ? UserAuth.get() : null;
    const session = (typeof UserAuth !== 'undefined') ?
UserAuth.isSessionActive() : false;
    slot.textContent = (u && session) ? `Logged in as ${u.email}` : '';
    if (logoutLi) logoutLi.classList.toggle('d-none', !(u && session));
  }
}
```

Description: Adds two items to the right side of the navbar: a status label and a logout button. It centers them vertically and shows or hides them based on whether a user profile exists and the session is active. Clicking on “Logout” clears the session and refreshes the status.