Project 3: Product Dashboard with DummyJSON

Overview

- **Goal**: Create a single-page web application that fetches product data from the **DummyJSON** API and displays it in a table. When a user clicks a row (or a button) for a product, show more details in a Bootstrap modal.
- **Technologies**:
  - **Bootstrap** (for layout and styling).
  - **Fetch API** (for HTTP requests).
  - **HTML/CSS/JavaScript** (no server-side code is required).

DummyJSON API

- **Base URL**: https://dummyjson.com/
- **Endpoint**: `/products` returns a JSON object with all products (by default, 30 products).
  - Example: `GET https://dummyjson.com/products`
  - The response includes a field `products` which is an array of product objects, each containing properties like `id`, `title`, `description`, `price`, `brand`, etc.

Requirements

1. **Fetch All Products**
   - Use the endpoint **`GET https://dummyjson.com/products`**.
   - Extract the `products` array from the JSON response.
   - Handle errors (e.g., network issues or unexpected server responses) by showing an alert or a user-friendly message.
2. **Show Products in a Bootstrap Table**
   - Display each product in **one row** of a table.
   - Include at least a few fields in the table, such as:
     - `id`
     - `title`
     - `price`
     - `brand`
   - You can add columns for `category` or `rating` if you like.
3. **Product Details in a Modal**
   - When the user selects (clicks) a product row (or a dedicated "View" button), open a **Bootstrap modal** that shows **additional details** of that product. For example:
     - `description`
     - `images[0]` (the first image, if you want to show a small thumbnail)
     - `stock`
   - The modal should be dynamically populated with the selected product's info.
4. **Error Handling and Promises**
   - Use the **`fetch()`** API in JavaScript to request data from the DummyJSON endpoint.
   - Handle **promise rejections** (e.g., using `.catch(...)`) to show an error message if something goes wrong with the request.

- Check `response.ok` and, if it's not, throw an error that can be handled in `.catch(...)`.

5. **Bootstrap Styling**
   - Use **Bootstrap** classes for a **responsive** layout.
   - At minimum, implement:
     - A **navbar** or a simple header for the page title ("Product Dashboard").
     - A **table** styled with `table table-striped` or `table-bordered`.
     - A **modal** component for product details.
   - Optionally, include a **loading spinner** while fetching data.

Suggested Steps to Implement

1. **Setup HTML and Bootstrap**
   - Create a file named `index.html`.
   - Include **Bootstrap CSS** and **Bootstrap JS** (or via CDN).
   - Add a `<div class="container my-4">` wrapper for your content.
   - Add a **table** with a `<thead>` (column headers) and an empty `<tbody>`.
2. **Add Modal in HTML**
   - At the bottom of `body`, include a **Bootstrap modal** skeleton. For example:

```
html
CopyEdit
<div class="modal fade" id="productModal" tabindex="-1" aria-
hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="productModalLabel">Product
Details</h5>
        <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-label="Close"></button>
      </div>
      <div class="modal-body">
        <!-- Product details go here -->
        <p id="productDescription"></p>
        <img id="productImage" alt="Product Image" class="img-
fluid d-none" />
      </div>
    </div>
  </div>
</div>
```

   - You'll **populate** this modal with details (e.g., product description, image) when the user selects a product.
3. **Fetch Data (JavaScript)**
   - Link a JavaScript file or include a `<script>` tag at the bottom of `index.html`.
   - Write a `fetchProducts()` function:

```
js
CopyEdit
async function fetchProducts() {
  try {
    const response = await
fetch('https://dummyjson.com/products');
    if (!response.ok) {
```

```
      throw new Error(`HTTP error! Status:
${response.status}`);
    }
    const data = await response.json();
    return data.products;  // Array of product objects
  } catch (error) {
    console.error('Error fetching products:', error);
    alert('Failed to fetch products. Please try again later.');
    return [];
  }
}
```

- You can also implement this without `async/await`, using `.then()` and `.catch()`. Either is fine.

4. **Render Table Rows**
   - After fetching the array of products, iterate over them to create table rows dynamically.
   - Suppose you have a `<tbody id="productTableBody">`. For each product:

```
js
CopyEdit
products.forEach(product => {
  const row = document.createElement('tr');

  // Basic cells
  row.innerHTML = `
    <td>${product.id}</td>
    <td>${product.title}</td>
    <td>${product.price}</td>
    <td>${product.brand}</td>
  `;

  // Optionally attach a click event to show the modal
  row.addEventListener('click', () => {
    showProductDetails(product);
  });

  document.getElementById('productTableBody').appendChild(row);
});
```

   - You could also add a dedicated **View** button in one column. When clicked, it calls `showProductDetails(product)`.

5. **Show Modal with Product Details**
   - Write a function `showProductDetails(product)` that:
     1. **Fills** the modal elements (e.g., `#productDescription`, `#productImage`) with data from the `product` object.
     2. **Displays** the modal using Bootstrap's JavaScript API:

```
js
CopyEdit
const modalLabel =
document.getElementById('productModalLabel');
modalLabel.textContent = product.title;

const desc =
document.getElementById('productDescription');
desc.textContent = product.description;
```

```
const img = document.getElementById('productImage');
if (product.images && product.images.length > 0) {
  img.src = product.images[0];
  img.classList.remove('d-none');
} else {
  img.classList.add('d-none');
}

// Show the modal
const myModal = new
bootstrap.Modal(document.getElementById('productModal'));
myModal.show();
```

- This ensures the user sees the product's **title, description,** and possibly an **image** in the modal.

6. **Main Initialization Flow**
   - In `script`, once the DOM is loaded (`window.onload` or `<script>` at the end of the body):

```
js
CopyEdit
window.onload = async function() {
  const products = await fetchProducts();
  if (products.length > 0) {
    // Render the table
    renderTable(products);
  }
};
```

   - This will **fetch** the data, then **render** it in the table.
7. **Error Handling**
   - If `fetchProducts()` throws an error, you **alert** the user and return an empty array.
   - Always check for `product.images` before accessing `[0]` to avoid errors if no images exist.
8. **Enhancements (Optional)**
   - **Search**: Add a search box to filter products by title or brand.
   - **Pagination**: DummyJSON supports queries like `?limit=10&skip=0` for partial results. You could show Next/Previous buttons for multiple pages.
   - **Loading Spinner**: Show a spinner while waiting for the fetch to complete, then hide it once the products load.

Deliverables

1. A single **HTML** page (`index.html`) that includes:
   - Bootstrap (CSS + JS).
   - A **table** with a header and body.
   - A **modal** to show product details.
2. A **JavaScript** section (or separate file) that:
   - Fetches data from `https://dummyjson.com/products`.

    - Handles any **errors** and displays a user-friendly message if something goes wrong.
    - Dynamically **creates** table rows and inserts them into the table body.
    - **Listens** for user clicks on a row/button to open the modal with product details.
3. Basic **CSS**/Bootstrap styling for a neat, responsive layout.

Tips for Success

- **Check** the network tab in your browser's developer tools to ensure the API call is working.
- **Console.log** the data to see the structure of each product object—so you know what fields to display.
- Always **test** the modal interaction and handle the case where a product might have no images.
- Remember that **`response.ok`** helps you detect if the HTTP status is in the success range (200-299). If not, `throw` an error and handle it in `.catch()` or in an `async/await` `try...catch`.