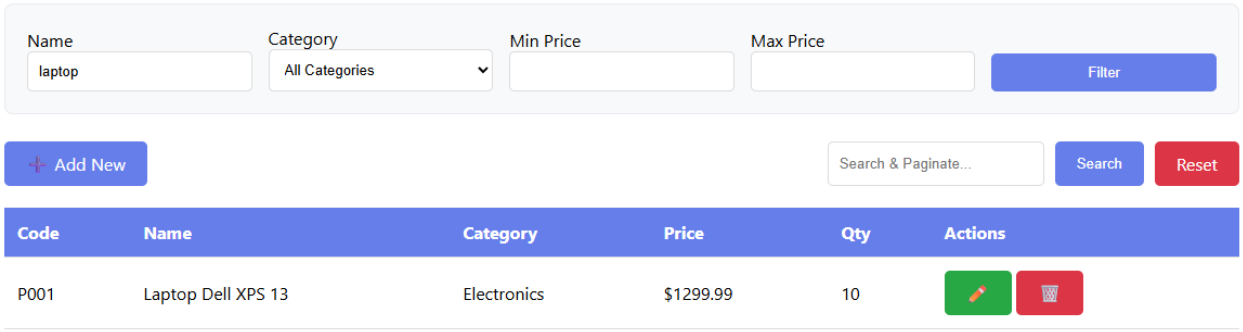
PART B: HOMEWORK EXERCISES

Name: Nguyễn Nhân Khang
ID: ITITWE22128

EXERCISE 5: ADVANCED SEARCH

Results:



Explain:

1. Multi-Criteria Search Flow (Advanced Search)
Goal: The user filters products by Name, Category, and Price Range.
Step 1: View (Interface) - User Interaction

- **Action:** The user enters "Laptop", selects Category "Electronics", enters a price range from 500 to 2000, and then clicks the "Filter" button.
- **Request:** The browser sends a `GET` request to the URL: `/products/advanced-search?name=Laptop&category=Electronics&minPrice=500&maxPrice=2000`

Step 2: Controller - Receiving Request

- **File:** `ProductController.java`
- **Method:** `advancedSearch(...)`
- **Processing:**
  - Receives parameters from the URL (`@RequestParam`).
  - Calls the **Service** to get the list of products satisfying the conditions.
  - Calls the **Service** to get the list of categories (for Task 5.2 - Dropdown).
  - Packages the results into the **Model**.

Java
```
// Controller calls Service
List<Product> products = productService.searchProducts(name, category, minPrice, maxPrice);
model.addAttribute("products", products); // Push data to view
```

Step 3: Service (Business Logic) - Intermediate Processing

- **File:** `ProductServiceImpl.java`
- **Method:** `searchProducts(...)`
- **Processing:**
  - Validates input data (e.g., converts empty strings to `null` so the query works correctly).
  - Calls the **Repository** to query the database.

Java
```
// Service handles small logic then calls Repo
```

```
if (name != null && name.trim().isEmpty()) name = null;
return productRepository.searchProducts(name, category, minPrice,
maxPrice);
```

Step 4: Repository (Data Layer) - Database Query

- **File:** `ProductRepository.java`
- **Method:** `searchProducts(...)`
- **Processing:** Executes the JPQL statement (Task 5.1) to filter data directly within the Database.

Java

```
// Repository executes Query
@Query("SELECT p FROM Product p WHERE ...")
List<Product> searchProducts(...);
```

Step 5: View (Response) - Display Results

- **File:** `product-list.html`
- **Processing:** Thymeleaf receives `products` from the Model and uses a `th:each` loop to re-render the data table.

---

2. Pagination Search Flow

Goal: The user searches quickly by name and views results page by page (Page 1, Page 2...).

Step 1: View (Interface)

- **Action:** The user enters the keyword "Dell" or clicks on page number "2" in the pagination bar.
- **Request:** The browser sends a `GET` request: `/products/search?keyword=Dell&page=1&size=10` (Note: page 1 in the code is index 0).

Step 2: Controller

- **Method:** `searchProductsPaginated(...)`
- **Processing:**
  - Creates a `Pageable` object from the `page` and `size` parameters.
  - Calls the **Service** with the keyword and the `Pageable` object.
  - Retrieves the `Page<Product>` result and total pages (`totalPages`) to put into the **Model**.

Java

```
// Create pagination object
Pageable pageable = PageRequest.of(page, size);
Page<Product> productPage =
productService.searchProductsPaginated(keyword, pageable);
```

Step 3: Service

- **Method:** `searchProductsPaginated(...)`
- **Processing:** Forwards the request to the Repository.

Java

```
return productRepository.findByNameContaining(keyword, pageable);
```

Step 4: Repository

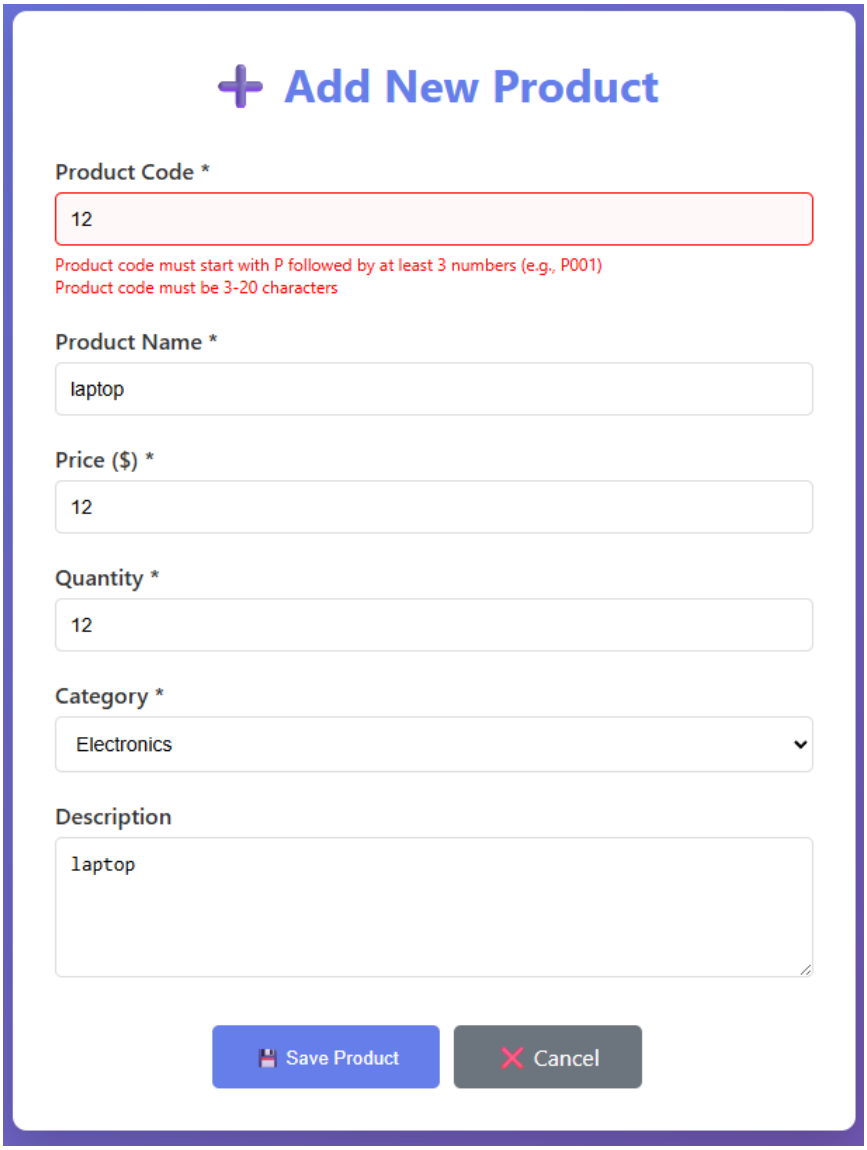
- **Method:** `findByNameContaining(String keyword, Pageable pageable)`
- **Processing:** Spring Data JPA automatically generates SQL statements with `LIMIT` and `OFFSET` (e.g., `LIMIT 10 OFFSET 0`) to retrieve exactly 10 products for the current page.

Step 5: View

- **Processing:**
  - Displays the list of products for the current page.
  - Based on `totalPages` and `currentPage`, draws the navigation bar (Previous, 1, 2, Next).

EXERCISE 6: VALIDATION

Result:



Explain:

1. Defining Rules (Model Layer)

Before the flow begins, the **Model** (`Product.java`) acts as the "lawyer," defining the rules:

- `productCode`: Cannot be empty, must start with 'P' + numbers.
- `price`: Must be > 0.01.
- `quantity`: Cannot be negative.

2. Handling Flow for Invalid Data (Validation Fail)

Example: The user enters `-10` for Price and clicks Save.

Step 1: View (Interface) - Sending Data

- The user fills out the form at `product-form.html` and clicks Submit.
- **Request:** A `POST /products/save` request is sent along with the form data (where `price = -10`).

Step 2: Controller - Receiving & Validating

- **File:** `ProductController.java`

- **Method:** `saveProduct(@Valid @ModelAttribute("product") Product product, BindingResult result, ...)`
- **Implicit Processing (Spring Magic):**
    1. The `@Valid` annotation triggers the validator (Hibernate Validator).
    2. It compares the submitted data (`price = -10`) against the rules in the Model (`@DecimalMin("0.01")`).
    3. **Error Detected** -> Spring automatically creates an error object and places it into the `BindingResult result`.

Step 3: Controller - Decision Making

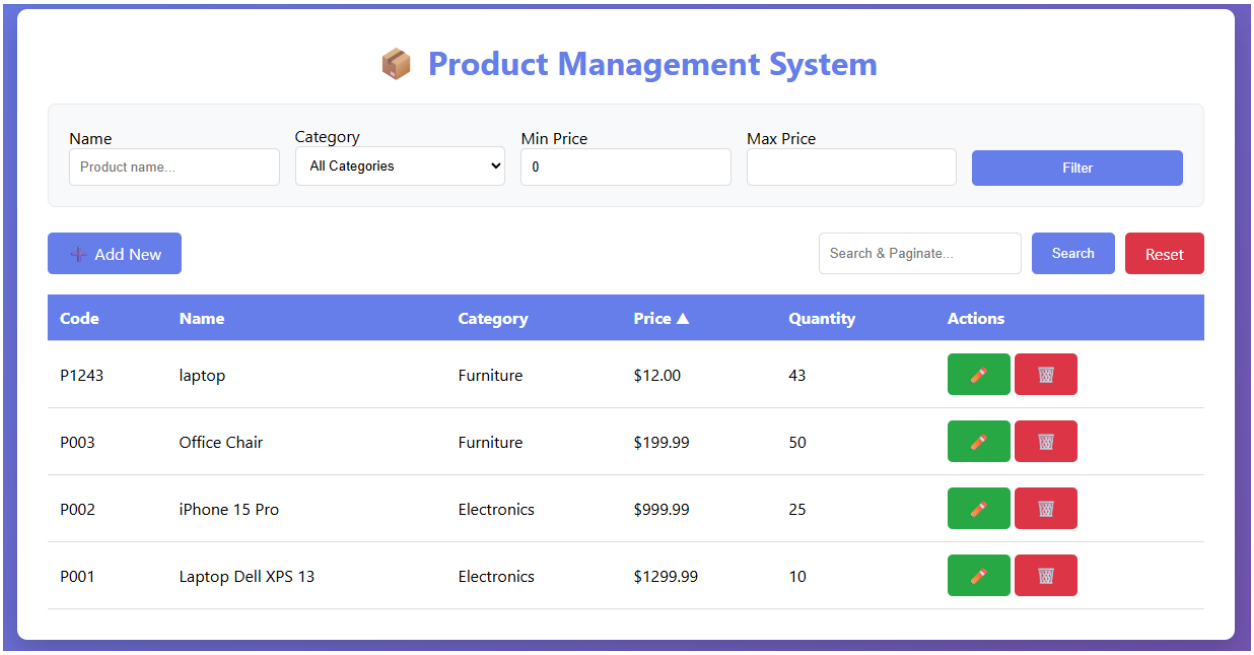
- The Controller executes the line: `if (result.hasErrors())`
- Since there are errors, this condition returns **TRUE**.
- **Action:** The Controller stops the saving process to the database. It immediately returns the view name: `return "product-form";` (Returning to the previous page).

Step 4: View (Interface) - Displaying Errors

- **File:** `product-form.html`
- Thymeleaf receives the `product` object (containing the invalid data) and the `BindingResult` (containing the error messages).
- **Processing:**
    1. `th:errorclass="error"`: Detects an error in the `price` field, adds the `.error` class to the input tag -> **Red border appears**.
    2. `th:errors="*{price}"`: Retrieves the error message "Price must be greater than 0" and displays it in the `<span>` tag.

EXERCISE 7: SORTING & FILTERING

Result:



Explain :

Step 1: View (Interface) - Generating a Smart Request Action:

- The user has entered `Category = Electronics` and clicked Filter (The URL is currently `...&category=Electronics...`).
- The user clicks on the **"Price"** column header in the result table.

Processing in View (`product-list.html`):

- Thymeleaf calculates the new URL based on `th:href`.
- It **retains** the parameter `category=Electronics`.
- It changes `sortBy=price`.
- It checks the current direction to **toggle** `sortDir` (e.g., from default to `asc`).

Request sent: `GET /products/advanced-search?category=Electronics&sortBy=price&sortDir=asc`

Step 2: Controller - Parameter Processing & Creating Sort Rules File: `ProductController.java` **Method:** `advancedSearch(...)`

Processing:

1. Receives all filter parameters (`category`) and sorting parameters (`sortBy`, `sortDir`).
2. **Key Logic:** Converts the `asc`/`desc` string into a Java `Sort` object.

Java

```
// Controller converts String -> Sort object
Sort sort = sortDir.equalsIgnoreCase("asc") ?
            Sort.by(sortBy).ascending() :
            Sort.by(sortBy).descending();

// Call Service with both filter and sort rules
List<Product> products = productService.searchProducts(name, category, ..., sort);
```

3. Repackages the `sortBy` and `sortDir` parameters into the `Model` to send back to the View (so the View knows how to draw the arrows and generate the link for the next click).

Step 3: Service (Middleware) - Forwarding File: `ProductServiceImpl.java` **Method:** `searchProducts(..., Sort sort)`

Processing: Simply passes the `Sort` object down to the Repository layer along with the filter parameters.

Step 4: Repository (Data Layer) - Dynamic Query Execution File: `ProductRepository.java` **Method:** `searchProducts(..., Sort sort)`

Mechanism (Spring Data Magic):

- You wrote the `@Query` JPQL for filtering (`WHERE p.category = :category ...`).
- Because you passed the `Sort` parameter at the end of the method, Spring Data JPA will **automatically append** the `ORDER BY` clause to the final SQL statement.

Actual Generated SQL:

SQL

```
SELECT * FROM products WHERE category = 'Electronics' ORDER BY price ASC;
```

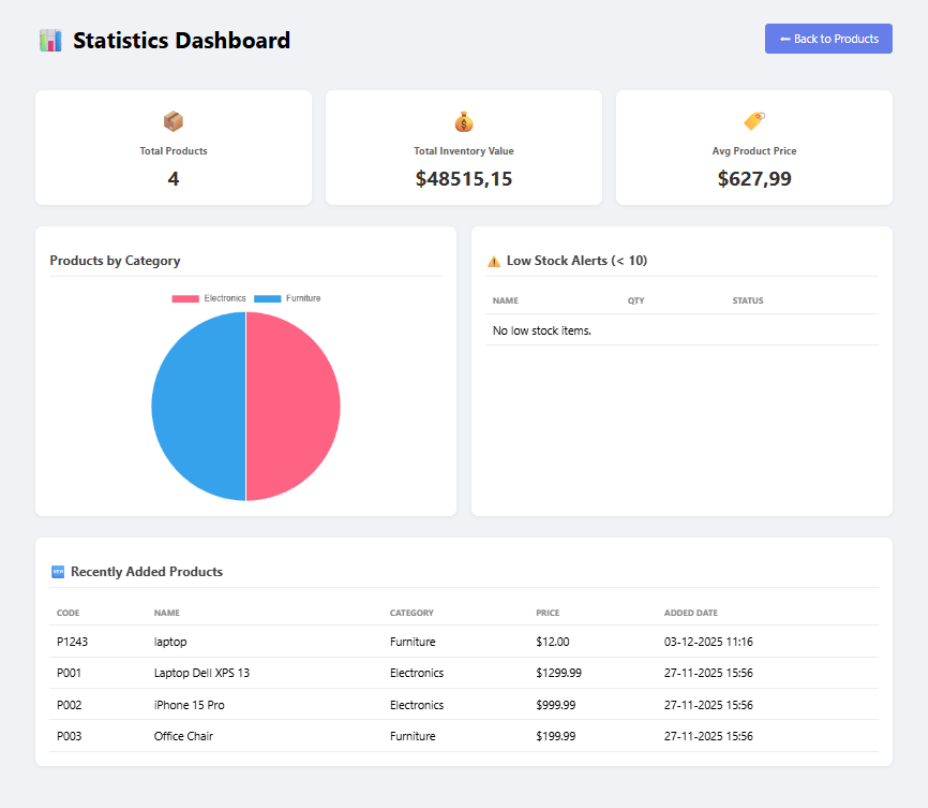
Result: Returns a list of electronics products, sorted by price from low to high.

Step 5: View (Interface) - Display & Status Update File: `product-list.html`

Display Processing:

1. The `th:each` loop displays the sorted list of products.
2. **Update Column Headers:**
    - **At the Price column:** Check `sortBy == 'price'`. True -> Display arrow ▲ (since `sortDir == 'asc'`).
    - **Update the link in the `<a>` tag of the Price column:** The next click will set `sortDir` to `desc`.
    - **Important:** Other columns (Name, Category) also update their links to **retain** the `category=Electronics` filter if the user switches to sorting by Name.

EXERCISE 8: STATISTICS DASHBOARD
Results:



Results:
Step 1: View (Interface) - Sending the Request

- **Action:** The user clicks on the `<a>` tag that you just added to `product-list.html`.
- **Request:** The browser sends a `GET` Request to the URL: `/dashboard`.

Step 2: Controller - Data Aggregation

- **File:** `DashboardController.java`
- **Method:** `showDashboard(Model model)`
- **Task:** Acts as a "collector." Instead of calling a single function, it calls multiple functions from the Service to gather all necessary data fragments.

Java

```
// The Controller repeatedly calls the Service to retrieve different
metrics
model.addAttribute("totalProducts",
productService.getAllProducts().size());
model.addAttribute("totalValue",
productService.calculateTotalValue());
// ... retrieve low stock alerts, recent products ...
```

- **Special processing for the Chart:** The Controller splits the category data into two separate lists (`categories` and `categoryCounts`) so the JavaScript library (Chart.js) can read them easily.

Step 3: Service (Business Logic) - Middleware

- **File:** `ProductServiceImpl.java`

- **Task:** Forwards calculation requests down to the Repository. Handles `null` cases (e.g., if there are no products yet, the total value must be `0` instead of throwing an error).

Java
```
public BigDecimal calculateTotalValue() {
    BigDecimal total = productRepository.calculateTotalValue();
    return total != null ? total : BigDecimal.ZERO; // Handle null
}
```

Step 4: Repository (Data Layer) - SQL Calculation

- **File:** `ProductRepository.java`
- **Task:** Executes Aggregation statements directly within the database. This is the most optimal approach (the Database performs calculations faster than Java).
- **Executed Queries:**
  - `SELECT SUM(p.price * p.quantity)...` -> Calculates total inventory value.
  - `SELECT AVG(p.price)...` -> Calculates average price.
  - `SELECT COUNT(p)...` -> Counts quantity by category.
  - `SELECT ... WHERE quantity < 10` -> Filters low stock items.
  - `SELECT ... ORDER BY created_at DESC` -> Retrieves the 5 most recent items.

Step 5: Controller - Packaging the Model

- After collecting 5-6 different types of data from the Service, the Controller packs everything into the `Model` object.
- **Result:** The `Model` now contains a rich "feast" of data: integers, decimals, lists of strings, lists of product objects, etc.
- **Navigation:** Returns the view: `return "dashboard";`

Step 6: View (Interface) - Displaying & Drawing Charts

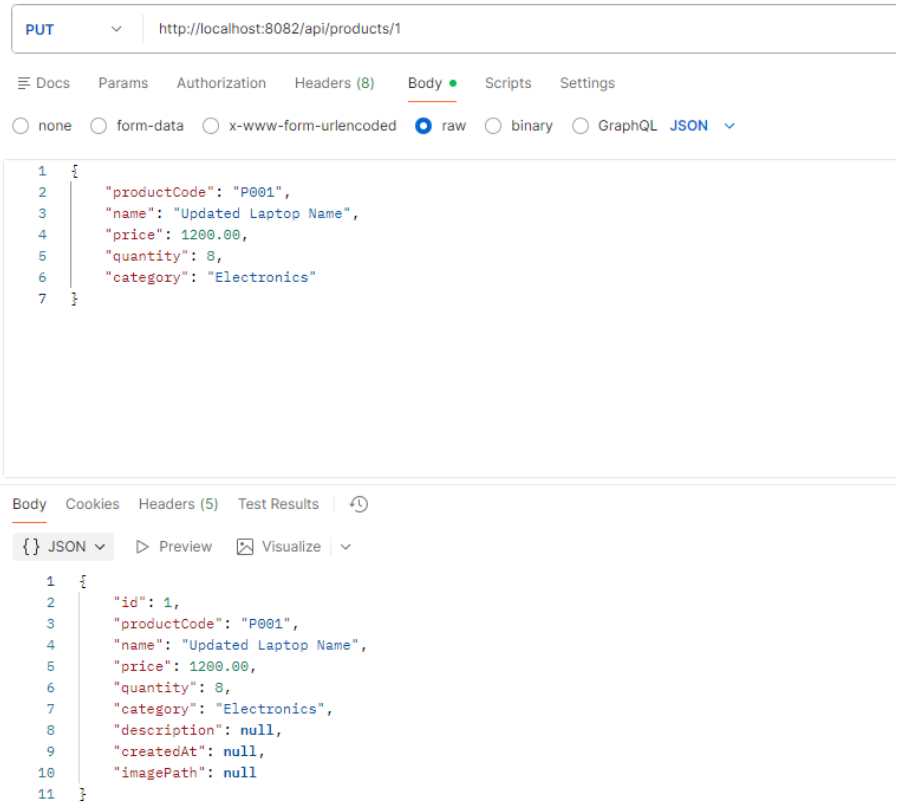
- **File:** `dashboard.html`
- **Thymeleaf Processing (Server-side):**
  - Fills numbers into the Cards (Total Value, Avg Price).
  - Creates HTML tables for Low Stock and Recent Products.
- **JavaScript Processing (Client-side):**
  - This is the most interesting step. Thymeleaf "injects" data from Java into JavaScript variables directly within the `<script>` tag:

JavaScript
```
var categories = /*[[${categories}]]*/ []; // Thymeleaf fills the
category list here
var counts = /*[[${categoryCounts}]]*/ []; // Thymeleaf fills the
quantity list here
```

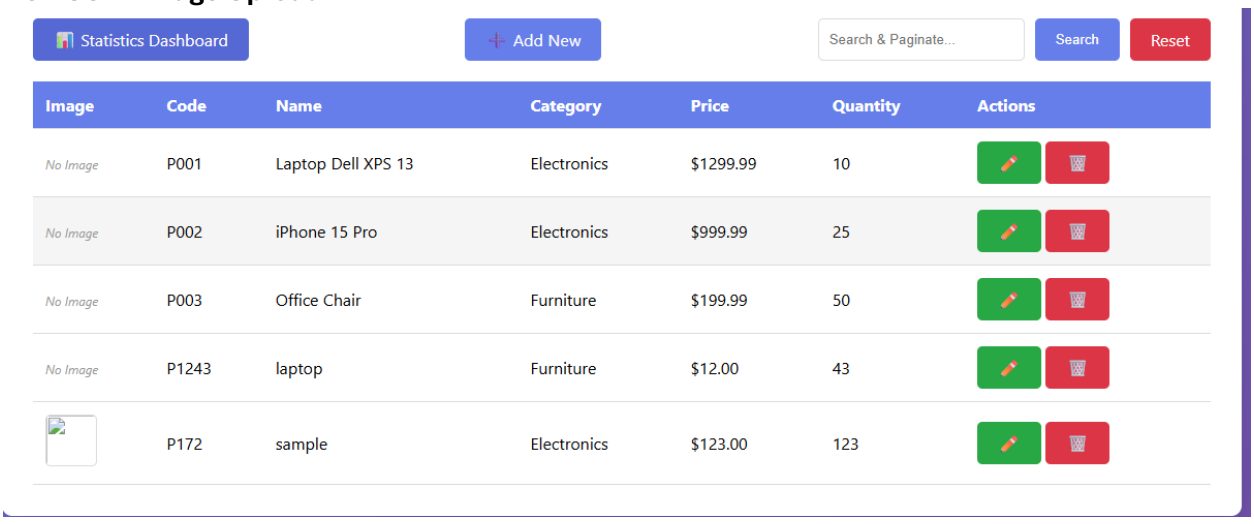
- The browser then uses the **Chart.js** library to read these two array variables and render a beautiful pie chart.

BONUS EXERCISES

BONUS 1: REST API Endpoints



BONUS 2: Image Upload



BONUS 3: Export to Excel (6 points)