1. Advanced Search, Pagination & Sorting (Ex 5 & 7)

This feature focuses on flexible data retrieval, allowing users to search by multiple criteria and sort the results.

File: `ProductRepository.java`

The core logic lies in the custom JPQL query. The `(:param IS NULL OR ...)` technique allows for optional filters.

```
@Query("SELECT p FROM Product p WHERE " +

       "(:name IS NULL OR p.name LIKE %:name%) AND " +          // 1. Skip if null, else partial match

       "(:category IS NULL OR p.category = :category) AND " +    // 2. Exact match for category

       "(:minPrice IS NULL OR p.price >= :minPrice) AND " +      // 3. Price lower bound

       "(:maxPrice IS NULL OR p.price <= :maxPrice)")            // 4. Price upper bound

Page<Product> searchProducts(@Param("name") String name, ... Pageable pageable);
```

File: `ProductController.java`

The `listProducts` method handles the request, processing both search filters and sorting parameters.

```
public String listProducts(..., String sortBy, String sortDir, ...) {

    // 1. Handle Sorting

    Sort sort = sortDir.equalsIgnoreCase("asc") ? Sort.by(sortBy).ascending() : Sort.by(sortBy).descending();



    // 2. Create Pageable object

    Pageable pageable = PageRequest.of(page, size, sort);



    // 3. Call search or find all

    if (hasFilters) {
```

```
        productService.searchProducts(name, category, minPrice, maxPrice, pageable);
    } else {
        productService.searchProducts(null, null, null, null, pageable);
    }
}
```

2. Validation (Ex 6)

Ensuring data integrity by validating user input before persistence.

File: `Product.java` (Entity)

Validation rules are defined using Jakarta Validation annotations directly on the entity fields.

```
@NotBlank(message = "Product code is required")
@Pattern(regexp = "^P\\d{3,}$") // Starts with 'P' followed by 3+ digits
private String productCode;

@DecimalMin(value = "0.01")
private BigDecimal price;
```

File: `ProductController.java`

The `@Valid` annotation triggers the validation, and `BindingResult` captures any errors.

```
@PostMapping("/save")
public String saveProduct(@Valid @ModelAttribute Product product, BindingResult
result, ...) {
    if (result.hasErrors()) {
        return "product-form"; // Return to form to show errors
    }
    productService.saveProduct(product);
```

```
}
```

3. Dashboard & Statistics (Ex 8, 9, 10)

Calculating and displaying key business metrics using optimized database queries.

File: `ProductRepository.java`

Using JPQL for aggregation is efficient as it avoids loading all data into memory.

```
// Count by category
@Query("SELECT COUNT(p) FROM Product p WHERE p.category = :category")
long countByCategory(String category);


// Sum total value (Price * Quantity)
@Query("SELECT SUM(p.price * p.quantity) FROM Product p")
BigDecimal calculateTotalValue();


// Find recent products (Spring Data Magic)
List<Product> findTop5ByOrderByCreatedAtDesc();
```

File: `DashboardController.java`

A dedicated controller aggregates this data for the view.

```
@GetMapping
public String showDashboard(Model model) {
    model.addAttribute("totalValue", productService.getTotalValue());


    // Calculate stats for each category
    Map<String, Long> stats = new HashMap<>();
```

```
        for (String cat : productService.getAllCategories()) {
            stats.put(cat, productService.countProductsByCategory(cat));
        }
        model.addAttribute("categoryCounts", stats);

        model.addAttribute("lowStock", productService.getLowStockProducts(10));
        return "dashboard";
    }
```