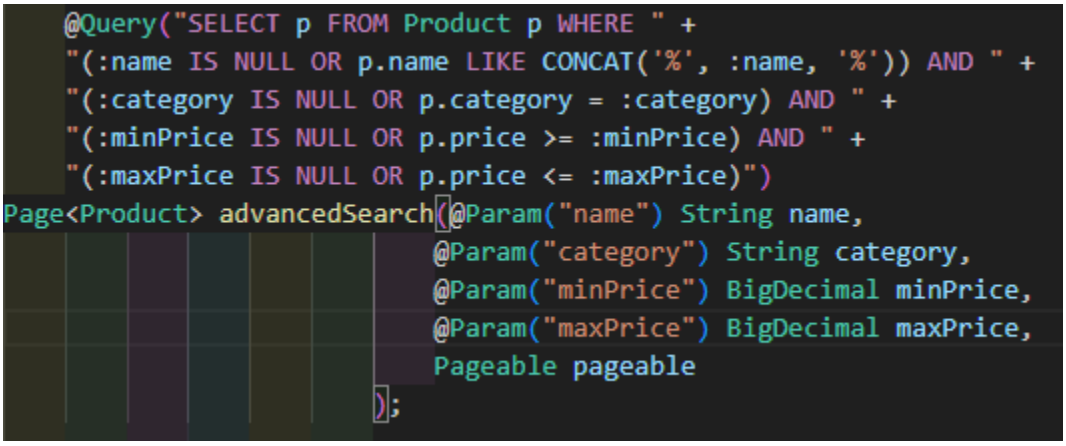
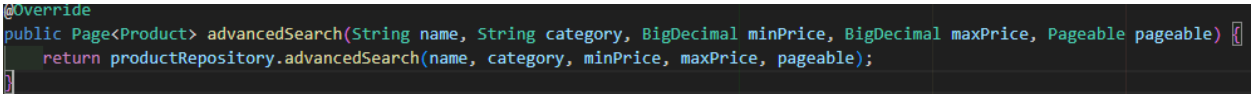
Task 5:

ProductRepository.java:




- Purpose: Performs a dynamic search on Product entities based on optional criteria.
- Query: Uses @Query with IS NULL OR ... to make each filter optional:
  + name → partial match (LIKE '%name%')
  + category → exact match
  + minPrice → products ≥ minPrice
  + maxPrice → products ≤ maxPrice
- Pagination: Returns results as a Page<Product> using Pageable.
- Flexibility: Any combination of filters can be applied; null values are ignored.

ProductServiceImpl.java:




- This method is in the service class and overrides an interface method.
- It calls the repository's advancedSearch method, passing all search parameters (name, category, minPrice, maxPrice) along with Pageable for pagination.
- Purpose: Acts as a bridge between the controller and repository, allowing controllers to fetch paginated search results without directly accessing the database.

ProductController.java:

```java
    @GetMapping("/advanced-search")
public String advancedSearch(
    @RequestParam(required = false) String name,
    @RequestParam(required = false) String category,
    @RequestParam(required = false) BigDecimal minPrice,
    @RequestParam(required = false) BigDecimal maxPrice,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "10") int size,
    Model model) {
    // Implementation
    if (name != null && name.isBlank()) name = null;
    if (category != null && category.isBlank()) category = null;
    Pageable pageable = PageRequest.of(page, size);
    Page<Product> products = productService.advancedSearch(name, category, minPrice, maxPrice, pageable);
    model.addAttribute("products", products.getContent());
    model.addAttribute("currentPage", products.getNumber());
    model.addAttribute("totalPages", products.getTotalPages());
    model.addAttribute("size", size);
    model.addAttribute("name", name);
    model.addAttribute("category", category);
    model.addAttribute("minPrice", minPrice);
    model.addAttribute("maxPrice", maxPrice);
    model.addAttribute("pageUrl", "/products/advanced-search");
    return "product-list";
    }
}
```

- Endpoint: GET /advanced-search
- Parameters: Optional search filters (name, category, minPrice, maxPrice) and pagination (page, size).
- Logic:
  + Converts blank strings to null so filters are ignored if empty.
  + Creates a Pageable object for pagination.
  + Calls the service's advancedSearch method to get a Page<Product>.
  + Adds search results and pagination info to the Model for the view.
- View: Returns "product-list" displaying filtered and paginated products.

```java
// Search products
@GetMapping("/search")
public String searchProducts(
    @RequestParam("keyword") String keyword,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "10") int size,
    Model model) {
        Pageable pageable = PageRequest.of(page, size);
        Page<Product> productPage = productService.searchProducts(keyword, pageable);
        model.addAttribute("products", productPage.getContent());
        model.addAttribute("currentPage", page);
        model.addAttribute("totalPages", productPage.getTotalPages());
        model.addAttribute("size", size);
        model.addAttribute("keyword", keyword);
        return "product-list";
}
```

- Endpoint: GET /search
- Parameters:
  + keyword → search term (required)
  + page, size → pagination (optional, default 0 and 10)
- Logic:
  + Creates a Pageable object for pagination.
  + Calls productService.searchProducts(keyword, pageable) to get matching products.
  + Adds products, pagination info, and keyword back to the Model.
- View: Returns "product-list" showing products that match the keyword with pagination support.

Task 6:

Product.java:

```java
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Product code is required")
    @Size(min = 3, max = 20, message = "Product code must be 3-20 characters")
    @Pattern(regexp = "^P\\d{3,}$", message = "Product code must start with P followed by numbers")
    private String productCode;

    @NotBlank(message = "Product name is required")
    @Size(min = 3, max = 100, message = "Name must be 3-100 characters")
    private String name;

    @NotNull(message = "Price is required")
    @DecimalMin(value = "0.01", message = "Price must be greater than 0")
    @DecimalMax(value = "999999.99", message = "Price is too high")
    private BigDecimal price;

    @NotNull(message = "Quantity is required")
    @Min(value = 0, message = "Quantity cannot be negative")
    private Integer quantity;

    @NotBlank(message = "Category is required")
    private String category;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;

    // Constructors
    public Product() {
    }
```

- @Entity & @Table(name = "products") → Maps this class to the products table in the database.
- @Id & @GeneratedValue(strategy = GenerationType.IDENTITY) → id is the primary key, auto-incremented by the database.
- productCode
  + Validations:
  - Must not be blank (@NotBlank)
  - Length 3–20 characters (@Size)
  - Must start with P followed by numbers (@Pattern)
- name
  + Validations: Not blank, length 3–100 characters.

- price
  + Must not be null, minimum 0.01, maximum 999999.99.
- quantity
  + Must not be null, cannot be negative.
- category → Cannot be blank.
- description → Stored as TEXT in the database.
- createdAt → Stores creation timestamp; cannot be updated.
- No-argument constructor → Required by JPA for entity instantiation.

ProductController.java:

```java
// Save product (create or update)
@PostMapping("/save")
public String saveProduct(
    @Valid @ModelAttribute("product") Product product, BindingResult result, Model model, RedirectAttributes redirectAttributes) {
        if (productService.isProductCodeTaken(product.getProductCode(), product.getId())) {
            result.rejectValue("productCode", "error.product", "Product code is already in use");
        }
        if (result.hasErrors()) {
            return "product-form";
        }
        try {
            productService.saveProduct(product);
            redirectAttributes.addFlashAttribute("message", "Product saved successfully!");
        } catch (Exception e) {
            redirectAttributes.addFlashAttribute("error", "Error: " + e.getMessage());
        }
    return "redirect:/products";
}
```

- Endpoint: POST /save – handles both creating a new product and updating an existing one.
- Parameters:
  + @Valid @ModelAttribute("product") Product product → Binds form data to a Product object and validates it based on the entity's annotations.
  + BindingResult result → Captures validation errors.
  + RedirectAttributes → Used to pass success/error messages after redirect.
- Logic:
  1. Check for duplicate productCode using the service; rejects the value if already in use.
  2. Validation: If any errors exist (result.hasErrors()), returns the form view (product-form).
  3. Save: Calls productService.saveProduct(product) to persist the product.
  4. Feedback: Adds a flash message for success or error.

- Redirect: After saving, redirects to /products to show the product list.

Task 7:

ProductController.java:

```java
// List all products
@GetMapping
public String listProducts(
    @RequestParam(required = false) String category,
    @RequestParam(required = false) String sortBy,
    @RequestParam(defaultValue = "asc") String sortDir,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "10") int size,
    Model model) {

        if (sortBy == null || sortBy.isBlank()) {
            sortBy = "id";
        }

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

        Pageable pageable = PageRequest.of(page, size, sort);
        Page<Product> productPage;

        if (category == null || category.isBlank() || category.equalsIgnoreCase(anotherString: "All")) {
            productPage = productService.getAllProducts(pageable);
        }
        else {
            productPage = productService.getProductsByCategory(category, pageable);
        }

        model.addAttribute("products", productPage.getContent());
        model.addAttribute("currentPage", productPage.getNumber());
        model.addAttribute("totalPages", productPage.getTotalPages());
        model.addAttribute("size", size);
        model.addAttribute("sortBy", sortBy);
        model.addAttribute("sortDir", sortDir);
        model.addAttribute("category", category);
        model.addAttribute("pageUrl", "/products");
        // Dynamic categories
        model.addAttribute("categories", productService.getAllCategories());
        return "product-list";  // Returns product-list.html
}
```

- Endpoint: GET /products – lists all products with optional category filtering, sorting, and pagination.
- Parameters:

+ category → Optional; filters products by category.
+ sortBy → Optional; field to sort by (defaults to "id").
+ sortDir → Optional; sort direction (asc or desc, default "asc").
+ page & size → Optional; pagination parameters.

- Logic:
  1. Determines sorting order with Sort.by(...).ascending()/descending().
  2. Creates a Pageable object for pagination and sorting.
  3.Fetches products:
  4. All products if no category or "All" is selected.
- Filtered by category otherwise.
- Adds products, pagination info, sorting info, selected category, and a list of dynamic categories to the Model.
- View: Returns "product-list" which displays products with pagination, sorting, and filtering.

Task 8:

ProductRepository.java:

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

@Query("SELECT SUM(p.price * p.quantity) FROM Product p")
BigDecimal calculateTotalValue();

@Query("SELECT AVG(p.price) FROM Product p")
BigDecimal calculateAveragePrice();

@Query("SELECT p FROM Product p WHERE p.quantity < :threshold")
List<Product> findLowStockProducts(@Param("threshold") int threshold);
```

- Returns the number of products in a specific category.
- Useful for category statistics or filtering summaries.

- Returns the average price of all products.
- Useful for reporting and analytics.
- Returns a list of products with quantity below a given threshold.
- Useful for inventory alerts or restocking notifications.

ProductController.java:

```java
@Controller
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductService productService;
    @GetMapping("/dashboard")
    public String showDashboard(Model model) {
        long totalProducts = productService.countProducts();
        BigDecimal totalValue = productService.calculateTotalValue();
        BigDecimal averagePrice = productService.calculateAveragePrice();
        List<Product> lowStock = productService.findLowStockProducts(threshold: 10);
        List<Product> recent = productService.findRecentProducts();
        Map<String, Long> categoryCounts = productService.countProductsByCategory();

        model.addAttribute("totalProducts", totalProducts);
        model.addAttribute("totalValue", totalValue);
        model.addAttribute("averagePrice", averagePrice);
        model.addAttribute("lowStockProducts", lowStock);
        model.addAttribute("recentProducts", recent);
        model.addAttribute("categoryCounts", categoryCounts);
        return "dashboard";
    }
```

- Endpoint: GET /dashboard – displays a summary of product statistics and insights.
- Logic:
  1. Fetches total number of products (countProducts).
  2.Calculates total inventory value (calculateTotalValue).
  3. Calculates average product price (calculateAveragePrice).
  4. Retrieves low-stock products (findLowStockProducts(10)).
  5. Retrieves recently added products (findRecentProducts).
  6. Counts products per category (countProductsByCategory).
- Model Attributes: Adds all the above data to the Model for rendering in the "dashboard" view.