Enterprise Product API Implementation Guide

Pagination + Sorting + Search + Category Filter (Spring Boot + Spring Data JPA)

Goal: Make the Product listing API production-ready by supporting pagination, sorting, keyword search, and filtering by categoryId.

Why interviewers love this topic: Because it shows you understand real-world API design and performance.

Features We Will Implement

- Pagination (page, size)
- Sorting (sortBy, sortDir)
- Search by product name keyword (case-insensitive)
- Filter products by categoryId
- Combine category filter + keyword search
- Return a standard paged response DTO with metadata

High-Level API Flow

1. Controller receives query parameters (page, size, sortBy, sortDir, keyword, categoryId).
2. Service builds Pageable object using PageRequest and Sort.
3. Repository executes correct query based on keyword/categoryId combination.
4. Service maps Product Entity to ProductResponseDto.
5. Service wraps data in PagedResponseDto (content + pagination metadata).
6. Controller wraps everything inside ApiResponse and returns to client.

STEP 1: Update ProductRepository

Why we do this: Spring Data JPA can auto-generate queries from method names. We create multiple methods to support search/filter combinations without writing SQL.

```
package com.advann.product_service.repository;

import com.advann.product_service.entity.Product;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository {

// Search by product name
Page findByNameContainingIgnoreCase(String keyword, Pageable pageable);

// Filter by category
Page findByCategoryId(Long categoryId, Pageable pageable);

// Search + category filter combined
Page findByCategoryIdAndNameContainingIgnoreCase(Long categoryId, String keyword,
Pageable pageable);
}
```

Interview Tip: Mention that returning **Page<T>** automatically provides total count, total pages, current page number, etc.

STEP 2: Create Pagination Response DTO

Why we do this: Returning Page object directly is not clean. We want a stable API response structure that frontend can easily consume.

```
package com.advann.product_service.dto;

import lombok.Builder;
import lombok.Data;
import java.util.List;

@Data
@Builder
public class PagedResponseDto {

private List content;
private int pageNumber;
private int pageSize;
private long totalElements;
private int totalPages;
private boolean last;
}
```

Interview Tip: This is called an API contract. Even if you change backend internally, frontend will not break.

STEP 3: Update ProductService Interface

Why we do this: Service layer should accept all parameters so controller remains thin. Business logic stays inside service.

```
PagedResponseDto getAllProducts(
int page,
int size,
String sortBy,
String sortDir,
String keyword,
Long categoryId
);
```

Note: keyword and categoryId are optional, so they can be null.

STEP 4: Implement Logic in ProductServiceImpl

Why we do this: Here we build Sort + Pageable and decide which repository query to call based on the input filters.

```
@Override
public PagedResponseDto getAllProducts(
int page,
int size,
String sortBy,
String sortDir,
String keyword,
Long categoryId
) {

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

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

Page productPage;

// category + keyword
if (categoryId != null && keyword != null && !keyword.isBlank()) {
productPage = productRepository.findByCategoryIdAndNameContainingIgnoreCase(categoryId,
keyword, pageable);
}
// only category filter
else if (categoryId != null) {
productPage = productRepository.findByCategoryId(categoryId, pageable);
}
// only keyword search
else if (keyword != null && !keyword.isBlank()) {
productPage = productRepository.findByNameContainingIgnoreCase(keyword, pageable);
}
// normal pagination
else {
productPage = productRepository.findAll(pageable);
}

List products = productPage.getContent()
.stream()
.map(product -> ProductResponseDto.builder()
.id(product.getId())
.name(product.getName())
.price(product.getPrice())
.quantity(product.getQuantity())
.categoryId(product.getCategory().getId())
.categoryName(product.getCategory().getName())
.build()
)
.toList();

return PagedResponseDto.builder()
.content(products)
.pageNumber(productPage.getNumber())
.pageSize(productPage.getSize())
.totalElements(productPage.getTotalElements())
.totalPages(productPage.getTotalPages())
.last(productPage.isLast())
.build();
```

```
}
```

Important: We never return Entity directly to client. We always map Entity to DTO.

Interview Tip: Mention that mapping protects sensitive fields and avoids infinite recursion issues in JPA relations.

STEP 5: Update ProductController API

Why we do this: Controller should only handle request/response. It should delegate business logic to service.

```
@GetMapping
public ResponseEntity>> getAllProducts(
@RequestParam(defaultValue = "0") int page,
@RequestParam(defaultValue = "5") int size,
@RequestParam(defaultValue = "id") String sortBy,
@RequestParam(defaultValue = "asc") String sortDir,
@RequestParam(required = false) String keyword,
@RequestParam(required = false) Long categoryId
) {

PagedResponseDto responseDto =
productService.getAllProducts(page, size, sortBy, sortDir, keyword, categoryId);

return ResponseEntity.ok(
ApiResponse.>builder()
.success(true)
.message("Products fetched successfully")
.data(responseDto)
.build()
);
}
```

Interview Tip: Default values ensure API works even if frontend does not send params.

STEP 6: Test APIs in Postman

Why we do this: Testing confirms pagination and filtering logic works correctly and metadata is returned.

Test Case	Request
Normal Pagination	GET /api/products?page=0&size=2
Sorting by Price Desc	GET /api/products?page=0&size=5&sortBy=price&sortDir=desc
Search Product	GET /api/products?keyword=iphone
Filter by Category	GET /api/products?categoryId=2
Category + Search Together	GET /api/products?categoryId=2&keyword=iphone

Common Interview Questions (With Strong Answers)

Q: Why do we use Pageable?

A: Pageable allows Spring Data JPA to apply limit/offset and sorting at DB level. It prevents loading huge datasets into memory.

Q: Why not return Page directly?

A: Returning Page directly exposes internal structure and makes API inconsistent. Using a custom DTO gives clean contract and better control.

Q: How does sorting work?

A: Sort is created using Sort.by(sortBy).ascending() or descending(). PageRequest combines pagination and sorting into a Pageable.

Q: How does keyword search work?

A: We use findByNameContainingIgnoreCase() which generates a LIKE query in SQL and ignores case.

Q: How do you combine filter and search?

A: We created a combined repository method findByCategoryIdAndNameContainingIgnoreCase() and call it when both inputs are present.

Extra Notes (Enterprise Points)

- Always validate page and size values (size should not be too large).
- For large systems, add indexes on name and category_id columns for faster search/filter.
- You can also implement default sorting (example: createdAt desc).
- This design is frontend friendly because it includes totalPages and totalElements.
- You can extend the same logic for multiple filters (price range, availability, brand, etc.).

Conclusion

Now your Product listing API is enterprise-ready. This is exactly how real projects implement scalable listing endpoints.