Part 7: Search and Filtering in your e-commerce API:

Step 1: Create DTOs for Products

1.1. Define DTOs

* Folder: DTOs
* ProductDTO.cs:

namespace AtirAPI.DTOs

{

public class ProductDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public int Stock { get; set; }

public int CategoryId { get; set; }

public CategoryDTO Category { get; set; }

}

public class CategoryDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

}

}

Step 2: Define Pagination DTO

2.1. Create PaginatedList DTO

* File: DTOs/PaginatedList.cs

namespace AtirAPI.DTOs

{

public class PaginatedList<T>

{

public int TotalItems { get; set; }

public int Page { get; set; }

public int PageSize { get; set; }

public IEnumerable<T> Items { get; set; }

}

}

Step 3: Update AutoMapper Configuration

3.1. AutoMapper Profile for Products

* File: Profiles/ProductProfile.cs

using AutoMapper;

using AtirAPI.Models;

using AtirAPI.DTOs;

namespace AtirAPI.Profiles

{

public class ProductProfile : Profile

{

public ProductProfile()

{

CreateMap<Product, ProductDTO>();

CreateMap<Category, CategoryDTO>();

}

}

}

* Ensure AutoMapper is configured in Program.cs:

builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

Step 4: Modify the Products Controller

4.1. Update ProductsController

* File: Controllers/ProductsController.cs

using AtirAPI.Data;

using AtirAPI.DTOs;

using AtirAPI.Models;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Linq;

namespace AtirAPI.Controllers

{

[Authorize]

[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{

private readonly AtirDbContext \_context;

private readonly IMapper \_mapper;

public ProductsController(AtirDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

// Existing methods...

// GET: api/Products/search

[HttpGet("search")]

public async Task<ActionResult<PaginatedList<ProductDTO>>> SearchProducts(

[FromQuery] string? name,

[FromQuery] int? categoryId,

[FromQuery] decimal? minPrice,

[FromQuery] decimal? maxPrice,

[FromQuery] int page = 1,

[FromQuery] int pageSize = 10,

[FromQuery] string? sortBy = null,

[FromQuery] bool ascending = true)

{

var query = \_context.Products.AsQueryable();

// Filter by name

if (!string.IsNullOrEmpty(name))

{

query = query.Where(p => p.Name.Contains(name));

}

// Filter by category

if (categoryId.HasValue)

{

query = query.Where(p => p.CategoryId == categoryId);

}

// Filter by price range

if (minPrice.HasValue)

{

query = query.Where(p => p.Price >= minPrice.Value);

}

if (maxPrice.HasValue)

{

query = query.Where(p => p.Price <= maxPrice.Value);

}

// Apply sorting

if (!string.IsNullOrEmpty(sortBy))

{

query = ascending

? query.OrderBy(p => EF.Property<object>(p, sortBy))

: query.OrderByDescending(p => EF.Property<object>(p, sortBy));

}

// Include category information and apply pagination

var totalItems = await query.CountAsync();

var products = await query

.Include(p => p.Category)

.Skip((page - 1) \* pageSize)

.Take(pageSize)

.ToListAsync();

var productDTOs = \_mapper.Map<List<ProductDTO>>(products);

return Ok(new PaginatedList<ProductDTO>

{

TotalItems = totalItems,

Page = page,

PageSize = pageSize,

Items = productDTOs

});

}

}

}

Step 5: Test the Search Endpoint

5.1. Using Postman or Swagger:

* Search by name: GET /api/Products/search?name=Smartphone
* Filter by category: GET /api/Products/search?categoryId=1
* Filter by price range: GET /api/Products/search?minPrice=100&maxPrice=500
* Combine filters: GET /api/Products/search?name=Phone&minPrice=100&maxPrice=1000&categoryId=1
* Pagination:
  + Page 1 with 5 items per page: GET /api/Products/search?page=1&pageSize=5
  + Page 2 with 10 items per page: GET /api/Products/search?page=2&pageSize=10
* Sorting:
  + Sort by price ascending: GET /api/Products/search?sortBy=Price&ascending=true
  + Sort by name descending: GET /api/Products/search?sortBy=Name&ascending=false

Step 6: Extend Filtering (Optional)

6.1. Additional Filters:

* Add more filters like stock availability or additional sorting options by modifying the SearchProducts method:

if (stockOnly)

{

query = query.Where(p => p.Stock > 0);

}

Add to the method signature:

[FromQuery] bool stockOnly = false

6.2. Test Extended Filters:

* Stock availability: GET /api/Products/search?stockOnly=true
* Additional sorting: Modify sortBy to accept more properties if needed.

Notes:

* Security: Validate the sortBy parameter to ensure it only contains allowed properties to prevent database injection vulnerabilities.
* Performance: For large datasets, consider using database indexing or full-text search capabilities.
* Scalability: If your product list grows significantly, look into using a dedicated search engine like Elasticsearch for more advanced search functionalities.

GET /api/products

**Bug fix**

The issue with self-referencing models in GET /api/Products endpoint

* Modify ProductsController for GetProducts:

[HttpGet]

public async Task<ActionResult<IEnumerable<ProductDTO>>> GetProducts()

{

var products = await \_context.Products

.Include(p => p.Category)

.ToListAsync();

return Ok(\_mapper.Map<IEnumerable<ProductDTO>>(products));

}

The same principle applies to the GET /api/Products/{id} endpoint.

Modify the GetProduct Method in ProductsController

* Change the return type and use AutoMapper to map to ProductDTO:

[HttpGet("{id}")]

public async Task<ActionResult<ProductDTO>> GetProduct(int id)

{

var product = await \_context.Products

.Include(p => p.Category)

.FirstOrDefaultAsync(p => p.Id == id);

if (product == null)

{

return NotFound();

}

return \_mapper.Map<ProductDTO>(product);

}