Part 4: Adding Categories with DTOs and AutoMapper

Objective: We'll expand our e-commerce API to include categories, using DTOs for data transfer to maintain clean, lean data contracts, and AutoMapper for simplifying object mappings. We'll also address issues like circular references and lazy loading.

1. Create the Category Entity

Add or modify the Models/Category.cs file:

namespace AtirAPI.Models

{

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public virtual ICollection<Product> Products { get; set; } = new List<Product>(); // Virtual for lazy loading

}

}

* Explanation:
  + Virtual: The Products property is declared as virtual to enable lazy loading. When lazy loading is enabled in Entity Framework Core, it can create proxy classes that override these virtual properties to load related data only when accessed, thus reducing unnecessary data loading.

2. Update the Product Entity

* File: Models/Product.cs

namespace AtirAPI.Models

{

public class Product

{

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 virtual Category Category { get; set; } // Virtual for lazy loading

}

}

* Explanation:
  + CategoryId: This is the foreign key that establishes the relationship with the Category table.
  + Virtual Category: The Category property is virtual for the same reason as in Category — to enable lazy loading of the category when needed.

3. Update the Database Context

* File: Data/AtirDbContext.cs

```

public DbSet<Category> Categories { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Category>()

.HasMany(c => c.Products)

.WithOne(p => p.Category)

.HasForeignKey(p => p.CategoryId);

base.OnModelCreating(modelBuilder);

}

```

* Explanation:
  + DbSet<Category>: Adds the Category entity to be managed by Entity Framework Core.
  + OnModelCreating: Configures the relationship between Category and Product, defining that one category can have many products, and each product belongs to one category via the CategoryId foreign key.

Add a migration and update the database:

dotnet ef migrations add AddCategoryEntity

dotnet ef database update

4. Create DTOs for Categories and Products

* Folder: DTOs
* CategoryCreateDTO.cs:

namespace AtirAPI.DTOs

{

public class CategoryCreateDTO

{

public string Name { get; set; }

public string Description { get; set; }

}

}

Explanation:

* + This DTO is used specifically for creating categories, containing only the data needed for creation.
* CategoryDTO.cs:

namespace AtirAPI.DTOs

{

public class CategoryDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

}

}

* Explanation:
  + This DTO is for returning category data, excluding any circular references or unnecessary details.
* CategoryUpdateDTO.cs:

namespace AtirAPI.DTOs

{

public class CategoryUpdateDTO

{

public string Name { get; set; }

public string Description { get; set; }

}

}

* ProductCreateDTO.cs:

namespace AtirAPI.DTOs

{

public class ProductCreateDTO

{

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; }

}

}

* Explanation:
  + Used for creating products, it includes the CategoryId to link the product to an existing category.
* 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; }

}

}

* Explanation:
  + This DTO is for returning product data with a reference to its category, avoiding direct entity exposure and circular references.

5. Configure AutoMapper

* Create Profiles/CategoryProfile.cs:

using AutoMapper;

using AtirAPI.Models;

using AtirAPI.DTOs;

namespace AtirAPI.Profiles

{

public class ProductProfile : Profile

{

public ProductProfile()

{

// Mapping for creating a product

CreateMap<ProductCreateDTO, Product>()

.ForMember(dest => dest.CategoryId, opt => opt.MapFrom(src => src.CategoryId))

.ForMember(dest => dest.Category, opt => opt.Ignore()); // Ignore Category since we only have the ID to map

// Mapping for returning a product

CreateMap<Product, ProductDTO>();

CreateMap<Category, CategoryDTO>();

}

}

}

* Explanation:
  + CreateMap<ProductCreateDTO, Product>: Maps from the DTO used for creation to the actual Product entity, focusing only on CategoryId since we can't directly map the Category object from this DTO.
  + CreateMap<Product, ProductDTO> and CreateMap<Category, CategoryDTO>: These mappings are for transforming database entities into DTOs for response, simplifying data structure and avoiding circular references.

6. Configure AutoMapper in Application Startup

* File: Program.cs

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

* Explanation:
  + This line registers AutoMapper with all assemblies in the application domain, allowing AutoMapper to find and use the Profile classes you've defined for mapping.

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

7. Create the Categories Controller

* File: Controllers/CategoriesController.cs

``` using AutoMapper;

using Microsoft.AspNetCore.Mvc;

using AtirAPI.Models;

using AtirAPI.DTOs;

using Microsoft.AspNetCore.Authorization;

using AtirAPI.Data;

using Microsoft.EntityFrameworkCore;

namespace ECommerceAPI.Controllers

{

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

[ApiController]

[Authorize]

public class CategoriesController : ControllerBase

{

private readonly AtirDbContext \_context;

private readonly IMapper \_mapper;

public CategoriesController(AtirDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

// GET : api/Categories

[HttpGet]

public async Task<ActionResult<IEnumerable<CategoryDTO>>> GetCategories()

{

var categories = await \_context.Categories.ToListAsync();

return Ok(\_mapper.Map<IEnumerable<CategoryDTO>>(categories));

}

// GET : api/Categories/5

[HttpGet("{id}")]

public async Task<ActionResult<CategoryDTO>> GetCategory(int id)

{

var category = await \_context.Categories.FindAsync(id);

if (category == null)

{

return NotFound();

}

return \_mapper.Map<CategoryDTO>(category);

}

// POST : api/Categories

[Authorize(Roles = "Admin")]

[HttpPost]

public async Task<ActionResult<CategoryDTO>> PostCategory(CategoryCreateDTO categoryDto)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

var category = \_mapper.Map<Category>(categoryDto);

\_context.Categories.Add(category);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetCategory), new { id = category.Id }, \_mapper.Map<CategoryDTO>(category));

}

// PUT : api/Categories/5

[Authorize(Roles = "Admin")]

[HttpPut("{id}")]

public async Task<IActionResult> PutCategory(int id, CategoryUpdateDTO categoryDto)

{

var category = await \_context.Categories.FindAsync(id);

if (category == null)

{

return NotFound();

}

\_mapper.Map(categoryDto, category); // Updates category with dto data

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!CategoryExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// DELETE: api/Categories/5

[Authorize(Roles = "Admin")]

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteCategory(int id)

{

var category = await \_context.Categories.FindAsync(id);

if (category == null)

{

return NotFound();

}

\_context.Categories.Remove(category);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool CategoryExists(int id)

{

return \_context.Categories.Any(e => e.Id == id);

}

}

}

```

* Explanation:
  + [Authorize]: Ensures that only authenticated users can access these endpoints.
  + IMapper: Injected into the controller to use AutoMapper for object mapping.
  + Methods: Each method uses DTOs for requests and responses, leveraging AutoMapper for type conversion, ensuring that only necessary data is transferred and avoiding direct entity exposure.

8. Update the Products Controller

* File: Controllers/ProductsController.cs

``` using AtirAPI.Data;

using AtirAPI.DTOs;

using AtirAPI.Models;

using AutoMapper;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

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;

}

// GET: api/Products

[HttpGet]

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

{

return await \_context.Products.Include(p => p.Category).ToListAsync();

}

// GET: api/Products/5

[HttpGet("{id}")]

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

{

var product = await \_context.Products.Include(p => p.Category).FirstOrDefaultAsync(p => p.Id == id);

if (product == null)

{

return NotFound();

}

return product;

}

[HttpPost]

public async Task<ActionResult<ProductDTO>> PostProduct([FromBody] ProductCreateDTO productDto)

{

var product = \_mapper.Map<Product>(productDto);

var category = await \_context.Categories.FindAsync(productDto.CategoryId);

if (category == null)

{

return BadRequest("The specified category does not exist.");

}

product.Category = category;

\_context.Products.Add(product);

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateException ex)

{

return StatusCode(500, "An error occurred while saving the product.");

}

var savedProduct = await \_context.Products

.Include(p => p.Category)

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

return CreatedAtAction(nameof(GetProduct), new { id = savedProduct.Id },

\_mapper.Map<ProductDTO>(savedProduct));

}

// PUT: api/Products/5

[HttpPut("{id}")]

public async Task<IActionResult> PutProduct(int id, Product product)

{

if (id != product.Id)

{

return BadRequest();

}

\_context.Entry(product).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ProductExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// DELETE: api/Products/5

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteProduct(int id)

{

var product = await \_context.Products.FindAsync(id);

if (product == null)

{

return NotFound();

}

\_context.Products.Remove(product);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool ProductExists(int id)

{

return \_context.Products.Any(e => e.Id == id);

}

}

}

```

* Explanation:
  + DTOs: Methods now use DTOs for both input and output, preventing circular references and controlling data exposure.
  + Mapping: AutoMapper is used to map between DTOs and entities, which simplifies code and ensures consistent data transfer.

9. Test the Endpoints

* Explanation:
  + Testing with tools like Postman or Swagger helps verify that the API endpoints work as expected with the new DTO and mapping setup. It also confirms that authorization roles are functioning correctly for admin actions.