Part 6:Order Management

Step 1: Create the Order and OrderItem Entities

1.1. Create Order Entity

* File: Models/Order.cs

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace AtirAPI.Models

{

public class Order

{

public int Id { get; set; }

[Required]

public int CustomerId { get; set; }

public virtual Customer Customer { get; set; }

public DateTime OrderDate { get; set; }

public decimal TotalAmount { get; set; }

public virtual ICollection<OrderItem> OrderItems { get; set; } = new List<OrderItem>();

}

}

1.2. Create OrderItem Entity

* File: Models/OrderItem.cs

namespace AtirAPI.Models

{

public class OrderItem

{

public int Id { get; set; }

public int OrderId { get; set; }

public virtual Order Order { get; set; }

public int ProductId { get; set; }

public virtual Product Product { get; set; }

public int Quantity { get; set; }

public decimal Price { get; set; }

}

}

Step 2: Update the Database Context

2.1. Add Orders and OrderItems to AtirDbContext

* File: Data/AtirDbContext.cs

using AtirAPI.Models;

using Microsoft.EntityFrameworkCore;

namespace AtirAPI.Data

{

public class AtirDbContext : DbContext

{

public AtirDbContext(DbContextOptions<AtirDbContext> options) : base(options) { }

// Existing DbSets...

public DbSet<Order> Orders { get; set; }

public DbSet<OrderItem> OrderItems { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// Existing configurations...

modelBuilder.Entity<Order>()

.HasMany(o => o.OrderItems)

.WithOne(oi => oi.Order)

.HasForeignKey(oi => oi.OrderId);

modelBuilder.Entity<OrderItem>()

.HasOne(oi => oi.Product)

.WithMany()

.HasForeignKey(oi => oi.ProductId);

base.OnModelCreating(modelBuilder);

}

}

}

2.2. Create Migration and Update Database

* Run these commands in your terminal:

dotnet ef migrations add AddOrderEntities

dotnet ef database update

Step 3: Create DTOs for Orders

3.1. Order DTOs

* Folder: DTOs
* OrderCreateDTO.cs:

namespace AtirAPI.DTOs

{

public class OrderCreateDTO

{

public int CustomerId { get; set; }

public ICollection<OrderItemCreateDTO> OrderItems { get; set; }

}

public class OrderItemCreateDTO

{

public int ProductId { get; set; }

public int Quantity { get; set; }

}

}

Step 4: Configure AutoMapper for Orders

4.1. Order Mapping Profile

* File: Profiles/OrderProfile.cs

using AutoMapper;

using AtirAPI.Models;

using AtirAPI.DTOs;

namespace AtirAPI.Profiles

{

public class OrderProfile : Profile

{

public OrderProfile()

{

CreateMap<OrderCreateDTO, Order>()

.ForMember(dest => dest.Customer, opt => opt.Ignore())

.ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems));

CreateMap<OrderItemCreateDTO, OrderItem>()

.ForMember(dest => dest.Order, opt => opt.Ignore())

.ForMember(dest => dest.Product, opt => opt.Ignore());

}

}

}

* Program.cs: Ensure AutoMapper is configured to include this profile:

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

Step 5: Create the Orders Controller

5.1. Orders Controller

* File: Controllers/OrdersController.cs

using AtirAPI.Data;

using AtirAPI.DTOs;

using AtirAPI.Models;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace AtirAPI.Controllers

{

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

[ApiController]

[Authorize]

public class OrdersController : ControllerBase

{

private readonly AtirDbContext \_context;

public OrdersController(AtirDbContext context)

{

\_context = context;

}

// GET: api/Orders

[HttpGet]

public async Task<ActionResult<IEnumerable<Order>>> GetOrders()

{

return await \_context.Orders

.Include(o => o.Customer)

.Include(o => o.OrderItems)

.ThenInclude(oi => oi.Product)

.ToListAsync();

}

// GET: api/Orders/5

[HttpGet("{id}")]

public async Task<ActionResult<Order>> GetOrder(int id)

{

var order = await \_context.Orders

.Include(o => o.Customer)

.Include(o => o.OrderItems)

.ThenInclude(oi => oi.Product)

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

if (order == null)

{

return NotFound();

}

return order;

}

// POST: api/Orders

[HttpPost]

public async Task<ActionResult<Order>> PostOrder(OrderCreateDTO orderDto)

{

// Validate customer existence

var customer = await \_context.Customers.FindAsync(orderDto.CustomerId);

if (customer == null)

{

return BadRequest("Customer not found.");

}

var order = new Order

{

CustomerId = orderDto.CustomerId,

OrderDate = DateTime.UtcNow,

OrderItems = new List<OrderItem>()

};

decimal totalAmount = 0;

foreach (var itemDto in orderDto.OrderItems)

{

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

if (product == null || product.Stock < itemDto.Quantity)

{

return BadRequest($"Insufficient stock for product ID {itemDto.ProductId}");

}

var orderItem = new OrderItem

{

ProductId = itemDto.ProductId,

Quantity = itemDto.Quantity,

Price = product.Price \* itemDto.Quantity,

Product = product // This will be ignored by EF when saving

};

order.OrderItems.Add(orderItem);

totalAmount += orderItem.Price;

product.Stock -= itemDto.Quantity; // Deduct stock

}

order.TotalAmount = totalAmount;

\_context.Orders.Add(order);

await \_context.SaveChangesAsync();

// Optionally, fetch the order with related data for return

var result = await \_context.Orders

.Include(o => o.Customer)

.Include(o => o.OrderItems)

.ThenInclude(oi => oi.Product)

.FirstOrDefaultAsync(o => o.Id == order.Id);

return CreatedAtAction(nameof(GetOrder), new { id = order.Id }, result);

}

// PUT and DELETE methods are left out for brevity. Implement similar to the GET and POST methods above.

private bool OrderExists(int id)

{

return \_context.Orders.Any(o => o.Id == id);

}

}

}

Step 6: Test the Endpoints

Use tools like Postman or Swagger to test:

* GET /api/Orders: Retrieves all orders with related data.
* GET /api/Orders/{id}: Retrieves a specific order by ID.
* POST /api/Orders: Creates a new order with:

{

"customerId": 1,

"orderItems": [

{

"productId": 1,

"quantity": 2

},

{

"productId": 2,

"quantity": 1

}

]

}

* Ensure to test for edge cases like insufficient stock or non-existent customers.

Step 7: Extend Functionality (Optional)

* Order Status: Add a status field in the Order model.
* Filtering: Implement query parameters for filtering orders.
* Payment Integration: Integrate a payment gateway.