Ideas for Jewel System BE - by M.Nhat & H.Nam

Developing a backend ASP.NET Web API for managing sale orders involves creating endpoints that handle the necessary CRUD (Create, Read, Update, Delete) operations and any other relevant actions related to sale orders. Here’s an outline of the steps and components you need to implement:

### Step-by-Step Guide for Developing the Backend API

#### 1. Set Up the ASP.NET Web API Project

1. \*\*Create a new ASP.NET Core Web API project\*\* using Visual Studio or the .NET CLI.

```sh

dotnet new webapi -n SalesOrderAPI

cd SalesOrderAPI

```

2. \*\*Add necessary packages\*\*:

- Entity Framework Core for database operations.

- Any other packages as needed (e.g., AutoMapper for object mapping).

```sh

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

```

#### 2. Define the Models

Create the models representing the entities involved in the sale order process. For simplicity, assume we have `Customer`, `Product`, `Order`, and `OrderItem`.

```csharp

public class Customer

{

public int Id { get; set; }

public string Name { get; set; }

public string Email { get; set; }

}

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int Stock { get; set; }

}

public class Order

{

public int Id { get; set; }

public DateTime OrderDate { get; set; }

public int CustomerId { get; set; }

public Customer Customer { get; set; }

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

}

public class OrderItem

{

public int Id { get; set; }

public int OrderId { get; set; }

public Order Order { get; set; }

public int ProductId { get; set; }

public Product Product { get; set; }

public int Quantity { get; set; }

public decimal UnitPrice { get; set; }

}

```

#### 3. Configure the DbContext

Create a `SalesOrderContext` to manage the database operations.

```csharp

public class SalesOrderContext : DbContext

{

public SalesOrderContext(DbContextOptions<SalesOrderContext> options)

: base(options)

{

}

public DbSet<Customer> Customers { get; set; }

public DbSet<Product> Products { get; set; }

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

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

}

```

Configure the connection string in `appsettings.json`:

```json

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=SalesOrderDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

```

Register the `SalesOrderContext` in `Startup.cs`:

```csharp

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<SalesOrderContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddControllers();

}

```

#### 4. Create the Repositories and Services

Abstract data access logic by creating repositories and services.

\*\*IRepository Interface\*\*:

```csharp

public interface IRepository<T> where T : class

{

Task<IEnumerable<T>> GetAllAsync();

Task<T> GetByIdAsync(int id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(int id);

}

```

\*\*Generic Repository\*\*:

```csharp

public class Repository<T> : IRepository<T> where T : class

{

private readonly SalesOrderContext \_context;

private readonly DbSet<T> \_dbSet;

public Repository(SalesOrderContext context)

{

\_context = context;

\_dbSet = context.Set<T>();

}

public async Task<IEnumerable<T>> GetAllAsync()

{

return await \_dbSet.ToListAsync();

}

public async Task<T> GetByIdAsync(int id)

{

return await \_dbSet.FindAsync(id);

}

public async Task AddAsync(T entity)

{

await \_dbSet.AddAsync(entity);

await \_context.SaveChangesAsync();

}

public async Task UpdateAsync(T entity)

{

\_dbSet.Update(entity);

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(int id)

{

var entity = await \_dbSet.FindAsync(id);

\_dbSet.Remove(entity);

await \_context.SaveChangesAsync();

}

}

```

\*\*OrderService\*\*:

```csharp

public class OrderService

{

private readonly IRepository<Order> \_orderRepository;

public OrderService(IRepository<Order> orderRepository)

{

\_orderRepository = orderRepository;

}

public async Task<IEnumerable<Order>> GetAllOrdersAsync()

{

return await \_orderRepository.GetAllAsync();

}

public async Task<Order> GetOrderByIdAsync(int id)

{

return await \_orderRepository.GetByIdAsync(id);

}

public async Task CreateOrderAsync(Order order)

{

await \_orderRepository.AddAsync(order);

}

public async Task UpdateOrderAsync(Order order)

{

await \_orderRepository.UpdateAsync(order);

}

public async Task DeleteOrderAsync(int id)

{

await \_orderRepository.DeleteAsync(id);

}

}

```

Register the repositories and services in `Startup.cs`:

```csharp

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<SalesOrderContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

services.AddScoped<OrderService>();

services.AddControllers();

}

```

#### 5. Create the Controllers

Create API controllers to handle HTTP requests for sale orders.

\*\*OrdersController\*\*:

```csharp

[ApiController]

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

public class OrdersController : ControllerBase

{

private readonly OrderService \_orderService;

public OrdersController(OrderService orderService)

{

\_orderService = orderService;

}

[HttpGet]

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

{

var orders = await \_orderService.GetAllOrdersAsync();

return Ok(orders);

}

[HttpGet("{id}")]

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

{

var order = await \_orderService.GetOrderByIdAsync(id);

if (order == null)

{

return NotFound();

}

return Ok(order);

}

[HttpPost]

public async Task<ActionResult<Order>> CreateOrder(Order order)

{

await \_orderService.CreateOrderAsync(order);

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

}

[HttpPut("{id}")]

public async Task<IActionResult> UpdateOrder(int id, Order order)

{

if (id != order.Id)

{

return BadRequest();

}

await \_orderService.UpdateOrderAsync(order);

return NoContent();

}

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteOrder(int id)

{

await \_orderService.DeleteOrderAsync(id);

return NoContent();

}

}

```

### Summary

This guide outlines the steps to create a backend API for managing sale orders in an ASP.NET Web API project. The key components include:

1. Setting up the project and configuring the DbContext.

2. Defining the necessary models.

3. Creating repositories and services for data access and business logic.

4. Implementing controllers to handle HTTP requests.

This setup provides a robust and maintainable architecture for your sale order management system. You can further expand it by adding authentication, authorization, validation, and additional features as needed.