.NET REST API Development: Building Maintainable Applications

1. Using Built-in Dependency Injection in ASP.NET Core

Dependency Injection (DI) is a design pattern that promotes loose coupling between components by providing dependencies from the outside rather than creating them internally. ASP.NET Core has a built-in DI container that allows you to register services and inject them throughout your application.

What is Dependency Injection?

Dependency Injection solves several problems:

- It decouples classes from their dependencies
- It makes testing easier through the use of mocks or stubs
- It allows for centralized configuration of services
- It provides a consistent way to manage object lifetimes

How to Use ASP.NET Core's DI Container

The DI container is configured in the `Program.cs` file (or `Startup.cs` in older versions). Here's a typical setup:

csharp

```csharp
var builder = WebApplication.CreateBuilder(args);

// Register services
builder.Services.AddControllers();
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddSingleton<ICacheService, RedisCacheService>();
builder.Services.AddTransient<IEmailSender, SmtpEmailSender>();

var app = builder.Build();

// Configure middleware
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Injecting Dependencies in Controllers

Once services are registered, you can inject them into controllers through constructor injection:

csharp

```csharp
public class OrdersController : ControllerBase
{
    private readonly IOrderService _orderService;
    private readonly ILogger<OrdersController> _logger;

    public OrdersController(IOrderService orderService, ILogger<OrdersController> logger)
    {
        _orderService = orderService;
        _logger = logger;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Order>>> GetOrders()
    {
        _logger.LogInformation("Getting all orders");
        var orders = await _orderService.GetAllOrdersAsync();
        return Ok(orders);
    }
}
```

Benefits of Using DI in ASP.NET Core

1. **Maintainability**: Classes are decoupled, making the codebase easier to maintain
2. **Testability**: Dependencies can be mocked during unit testing
3. **Flexibility**: Implementations can be swapped without changing the consuming code

4. **Lifetime Management**: The framework handles object creation and disposal

2. Understanding Service Lifetimes: AddScoped, AddSingleton, and AddTransient

ASP.NET Core's DI container offers three primary service lifetimes, each suitable for different scenarios.

AddTransient

Transient services are created each time they are requested from the service container.

csharp

```csharp
builder.Services.AddTransient<IEmailSender, SmtpEmailSender>();
```

Characteristics:

- A new instance is created for every controller and every service that depends on it
- Ideal for lightweight, stateless services
- Safest option, as it avoids sharing state between different components/requests

Example use case: Services that perform one-off operations like sending emails, validating input, or performing calculations.

csharp

```csharp
public class EmailSender : IEmailSender
{
    public async Task SendEmailAsync(string to, string subject, string body)
    {
        // Each call gets a fresh instance
        // No risk of sharing state between different email operations
        await SendEmailInternalAsync(to, subject, body);
    }
}
```

AddScoped

Scoped services are created once per client request (connection).

csharp

```csharp
builder.Services.AddScoped<IOrderService, OrderService>();
```

Characteristics:

- Same instance is used within a single HTTP request
- Different instance is created for different HTTP requests
- Useful for services that need to maintain state during a request

Example use case: Services that need to track information throughout a request, like shopping carts or database contexts.

csharp

```csharp
public class OrderService : IOrderService
{
    private readonly ApplicationDbContext _dbContext;

    public OrderService(ApplicationDbContext dbContext)
    {
        // DbContext is typically scoped - one instance per HTTP request
        _dbContext = dbContext;
    }

    public async Task<Order> CreateOrderAsync(OrderDto orderDto)
    {
        var order = new Order
        {
            CustomerId = orderDto.CustomerId,
            OrderDate = DateTime.UtcNow
        };

        _dbContext.Orders.Add(order);
        await _dbContext.SaveChangesAsync();
        return order;
    }
}
```

AddSingleton

Singleton services are created once for the lifetime of the application.

csharp

```
builder.Services.AddSingleton<ICacheService, RedisCacheService>();
```

Characteristics:

- Created when first requested or at app startup
- Same instance used throughout the entire application lifetime
- Shared across all HTTP requests and all users

Example use case: Services that maintain state across the application, such as caches, configuration settings, or connection pools.

csharp

```
public class RedisCacheService : ICacheService
{
    private readonly ConnectionMultiplexer _redisConnection;

    public RedisCacheService(IConfiguration configuration)
    {
        // Expensive connection is created only once for the entire application
        _redisConnection = ConnectionMultiplexer.Connect(configuration.GetConnectionString("Redis"));
    }

    public async Task<T> GetAsync<T>(string key)
    {
        var db = _redisConnection.GetDatabase();
        var value = await db.StringGetAsync(key);
        return value.IsNull ? default : JsonSerializer.Deserialize<T>(value);
    }
}
```

Choosing the Right Lifetime

Lifetime	When to Use	When to Avoid
Transient	Lightweight, stateless services	Services with expensive initialization
Scoped	Services that track state during a request, like DbContext	Services that need to be shared across requests
Singleton	Services shared across the application	Services with request-specific state or that store user data

Lifetime Scope Validation

ASP.NET Core can detect common DI lifetime mismatches:

csharp

```
builder.Services.AddSingleton<ISingletonService, SingletonService>();
// This will cause issues - a singleton depending on a scoped service
public class SingletonService : ISingletonService
{
    private readonly IScopedService _scopedService; // Problematic!

    public SingletonService(IScopedService scopedService)
    {
        _scopedService = scopedService;
    }
}
```

Important Rule: A service with a longer lifetime should not depend on a service with a shorter lifetime.

3. Using Injected Configuration from appsettings.json in Classes

ASP.NET Core provides a flexible configuration system that can pull values from multiple sources, with `appsettings.json` being the most common.

Configuration Setup

Configuration is typically set up in `Program.cs`:

csharp

```
var builder = WebApplication.CreateBuilder(args);

// Configuration is automatically set up with appsettings.json
// You can add additional configuration sources if needed
builder.Configuration.AddJsonFile("customsettings.json", optional: true);
```

Basic Configuration Injection

The simplest way to access configuration is by injecting `IConfiguration`:

csharp

```
public class EmailService : IEmailService
{
    private readonly string _smtpServer;
    private readonly int _smtpPort;
    private readonly string _senderEmail;

    public EmailService(IConfiguration configuration)
    {
        _smtpServer = configuration["EmailSettings:SmtpServer"];
        _smtpPort = int.Parse(configuration["EmailSettings:SmtpPort"]);
        _senderEmail = configuration["EmailSettings:SenderEmail"];
    }

    public async Task SendEmailAsync(string recipient, string subject, string body)
    {
        // Use the configuration values to send an email
    }
}
```

Strongly-Typed Configuration with Options Pattern

A better approach is to use the Options pattern, which provides strongly-typed access to configuration sections:

1. First, create a class that represents your configuration section:

csharp

```
public class EmailSettings
{
    public string SmtpServer { get; set; }
    public int SmtpPort { get; set; }
    public string SenderEmail { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
}
```

2. Register the configuration section in `Program.cs`:

csharp

```
builder.Services.Configure<EmailSettings>(
    builder.Configuration.GetSection("EmailSettings"));
```

3. Inject and use the typed options in your service:

csharp

```
public class EmailService : IEmailService
{
    private readonly EmailSettings _emailSettings;

    public EmailService(IOptions<EmailSettings> emailOptions)
    {
        _emailSettings = emailOptions.Value;
    }
    public async Task SendEmailAsync(string recipient, string subject, string body)
    {
        var client = new SmtpClient(_emailSettings.SmtpServer, _emailSettings.SmtpPort)
        {
            Credentials = new NetworkCredential(
                _emailSettings.Username,
                _emailSettings.Password)
        };

        var message = new MailMessage(
            _emailSettings.SenderEmail,
            recipient,
            subject,
            body);

        await client.SendMailAsync(message);
    }
}
```

Configuration in appsettings.json

The corresponding `appsettings.json` would look like this:

json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "EmailSettings": {
    "SmtpServer": "smtp.example.com",
    "SmtpPort": 587,
    "SenderEmail": "noreply@example.com",
    "Username": "apiuser",
    "Password": "this-should-be-in-user-secrets"
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=myserver;Database=mydb;User Id=myuser;Password=mypassword;"
  },
  "AllowedHosts": "*"
}
```

Accessing Configuration in Different Environments

ASP.NET Core loads environment-specific configuration files:

- `appsettings.json`
- `appsettings.{Environment}.json` (e.g., `appsettings.Development.json`)

The environment-specific file overrides settings in the base file:

csharp

```
// appsettings.Development.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "Microsoft.AspNetCore": "Information"
    }
  },
  "EmailSettings": {
    "SmtpServer": "localhost",
    "SmtpPort": 25
  }
}
```

4. Working with User Secrets

User Secrets provide a secure way to store sensitive configuration data during development, keeping it out of source control.

Why Use User Secrets?

- Prevents sensitive data from being committed to source control
- Reduces the risk of accidentally exposing credentials
- Separates sensitive configuration from application code

Setting Up User Secrets

1. Initialize the user secrets store for your project:

bash

```bash
dotnet user-secrets init --project YourProjectName.csproj
```

This adds a `UserSecretsId` element to your project file:

xml

```xml
<PropertyGroup>
  <TargetFramework>net7.0</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

2. Add secrets to the store:

bash

```bash
dotnet user-secrets set "EmailSettings:Password" "your-secure-password" --project YourProjectName.csproj
dotnet user-secrets set "ConnectionStrings:DefaultConnection" "Server=myserver;Database=mydb;User
Id=myuser;Password=mysecurepassword;" --project YourProjectName.csproj
```

3. Access secrets just like normal configuration:

csharp

```csharp
// The same code works whether the setting comes from appsettings.json or user secrets
public class DatabaseService : IDatabaseService
{
    private readonly string _connectionString;

    public DatabaseService(IConfiguration configuration)
    {
        _connectionString = configuration.GetConnectionString("DefaultConnection");
    }
}
```

User Secrets Location

User secrets are stored on the local development machine:

- Windows: `%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json`
- macOS/Linux: `~/.microsoft/usersecrets/<user_secrets_id>/secrets.json`

User Secrets JSON Format

The secrets are stored in a JSON file:

```json
{
  "EmailSettings:Password": "your-secure-password",
  "ConnectionStrings:DefaultConnection": "Server=myserver;Database=mydb;User Id=myuser;Password=mysecurepassword;"
}
```

Best Practices for User Secrets

1. Only use user secrets for development environments
2. For production, use environment variables or a secure configuration provider like Azure Key Vault
3. Never check in the `secrets.json` file or share the `UserSecretsId`
4. Don't store truly sensitive information (like production credentials) in user secrets
5. Document which secrets developers need to set up locally

5. Horizontal Layered Architecture in ASP.NET Core REST APIs

Layered architecture is a common design pattern for organizing code in enterprise applications. It divides an application into horizontal layers, where each layer has a specific responsibility and communicates with adjacent layers using well-defined interfaces. This approach provides separation of concerns and makes the application easier to maintain, test, and scale.

5.1 Understanding Layered Architecture

A traditional layered architecture for an ASP.NET Core REST API typically consists of the following layers:

1. **Presentation Layer** (Controllers)
   - Handles HTTP requests and responses
   - Validates input data
   - Maps DTOs to domain models and vice versa
   - Manages authentication and authorization
2. **Service Layer** (Business Logic)
   - Implements business rules and workflows
   - Orchestrates operations that span multiple repositories
   - Handles transactions and ensures data consistency
   - Performs validation that requires business knowledge
3. **Repository Layer** (Data Access)
   - Abstracts the data access logic
   - Handles CRUD operations
   - Communicates with databases, external APIs, or file systems
   - Maps database entities to domain models
4. **Domain Layer** (Models)
   - Contains domain models and business entities
   - Defines interfaces for repositories and services
   - Encapsulates business rules specific to individual entities

5.2 Project Structure Example

A typical project structure for a layered ASP.NET Core API might look like this:

```
YourSolution/
├── YourSolution.API/              (Presentation Layer)
│   ├── Controllers/
│   ├── DTOs/
│   ├── Filters/
│   ├── Middleware/
│   ├── Program.cs
│   └── appsettings.json
│
├── YourSolution.Services/         (Service Layer)
│   ├── Interfaces/
│   │   └── IOrderService.cs
│   ├── Implementations/
│   │   └── OrderService.cs
│   ├── DTOs/
│   │   └── OrderServiceDto.cs
│   └── Validators/
│       └── OrderValidator.cs
│
├── YourSolution.Domain/           (Domain Layer)
│   ├── Entities/
│   │   └── Order.cs
│   ├── Interfaces/
│   │   └── IOrderRepository.cs
│   ├── Enums/
│   └── Exceptions/
│
├── YourSolution.Infrastructure/   (Repository Layer)
│   ├── Data/
│   │   ├── ApplicationDbContext.cs
│   │   └── Configurations/
│   ├── Repositories/
│   │   └── OrderRepository.cs
│   ├── External/
│   │   └── PaymentGatewayClient.cs
│   └── Migrations/
│
└── YourSolution.Common/              (Shared Components)
    ├── Extensions/
    ├── Constants/
    └── Utilities/
```

5.3 Code Examples

Let's look at a complete example of an Order management system implemented with layered architecture.

Domain Layer

Start with the domain entities and repository interfaces:

csharp

```csharp
// YourSolution.Domain/Entities/Order.cs
namespace YourSolution.Domain.Entities
{
    public class Order
    {
        public int Id { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public decimal TotalAmount { get; set; }
        public string CustomerId { get; set; }
        public OrderStatus Status { get; set; }
        public List<OrderItem> Items { get; set; } = new List<OrderItem>();
    }

    public class OrderItem
    {
        public int Id { get; set; }
        public int OrderId { get; set; }
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public int Quantity { get; set; }
    }

    public enum OrderStatus
    {
        Pending,
        Processing,
        Shipped,
        Delivered,
        Cancelled
    }
}

// YourSolution.Domain/Interfaces/IOrderRepository.cs
namespace YourSolution.Domain.Interfaces
{
    public interface IOrderRepository
    {
        Task<Order> GetByIdAsync(int id);
        Task<Order> GetByOrderNumberAsync(string orderNumber);
        Task<IEnumerable<Order>> GetAllAsync();
        Task<IEnumerable<Order>> GetByCustomerIdAsync(string customerId);
        Task<Order> CreateAsync(Order order);
        Task UpdateAsync(Order order);
        Task DeleteAsync(int id);
    }
}
```

Repository Layer

Next, implement the repository interface:

csharp

```
// YourSolution.Infrastructure/Data/ApplicationDbContext.cs
namespace YourSolution.Infrastructure.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Order>()
                .HasMany(o => o.Items)
                .WithOne()
                .HasForeignKey(i => i.OrderId);

            modelBuilder.Entity<Order>()
                .Property(o => o.TotalAmount)
                .HasColumnType("decimal(18,2)");

            modelBuilder.Entity<OrderItem>()
                .Property(i => i.UnitPrice)
                .HasColumnType("decimal(18,2)");
        }
    }
}

// YourSolution.Infrastructure/Repositories/OrderRepository.cs
namespace YourSolution.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
        private readonly ApplicationDbContext _context;

        public OrderRepository(ApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<Order> GetByIdAsync(int id)
        {
            return await _context.Orders
                .Include(o => o.Items)
                .FirstOrDefaultAsync(o => o.Id == id);
        }

        public async Task<Order> GetByOrderNumberAsync(string orderNumber)
        {
            return await _context.Orders
                .Include(o => o.Items)
                .FirstOrDefaultAsync(o => o.OrderNumber == orderNumber);
        }

        public async Task<IEnumerable<Order>> GetAllAsync()
        {
            return await _context.Orders
                .Include(o => o.Items)
                .ToListAsync();
        }

        public async Task<IEnumerable<Order>> GetByCustomerIdAsync(string customerId)
        {
            return await _context.Orders
                .Include(o => o.Items)
                .Where(o => o.CustomerId == customerId)
                .ToListAsync();
        }

        public async Task<Order> CreateAsync(Order order)
        {
            _context.Orders.Add(order);
            await _context.SaveChangesAsync();
            return order;
        }
        public async Task UpdateAsync(Order order)
        {
            _context.Entry(order).State = EntityState.Modified;

            foreach (var item in order.Items)
            {
                if (item.Id == 0)
                    _context.OrderItems.Add(item);
                else
                    _context.Entry(item).State = EntityState.Modified;
            }

            await _context.SaveChangesAsync();
        }

        public async Task DeleteAsync(int id)
        {
            var order = await _context.Orders.FindAsync(id);
            if (order != null)
            {
                _context.Orders.Remove(order);
                await _context.SaveChangesAsync();
            }
        }
    }
}
```

Service Layer

Now, implement the service interface and implementation:

csharp

```
// YourSolution.Services/DTOs/OrderDto.cs
namespace YourSolution.Services.DTOs
{
    public class OrderDto
    {
        public int Id { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public decimal TotalAmount { get; set; }
        public string CustomerId { get; set; }
        public string Status { get; set; }
        public List<OrderItemDto> Items { get; set; } = new List<OrderItemDto>();
    }

    public class OrderItemDto
    {
        public int Id { get; set; }
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public int Quantity { get; set; }
        public decimal Subtotal => UnitPrice * Quantity;
    }

    public class CreateOrderDto
    {
        public string CustomerId { get; set; }
        public List<CreateOrderItemDto> Items { get; set; } = new List<CreateOrderItemDto>();
    }

    public class CreateOrderItemDto
    {
        public int ProductId { get; set; }
        public int Quantity { get; set; }
    }
}

// YourSolution.Services/Interfaces/IOrderService.cs
namespace YourSolution.Services.Interfaces
{
    public interface IOrderService
    {
        Task<OrderDto> GetOrderByIdAsync(int id);
        Task<OrderDto> GetOrderByNumberAsync(string orderNumber);
        Task<IEnumerable<OrderDto>> GetAllOrdersAsync();
        Task<IEnumerable<OrderDto>> GetOrdersByCustomerIdAsync(string customerId);
        Task<OrderDto> CreateOrderAsync(CreateOrderDto createOrderDto);
        Task UpdateOrderStatusAsync(int id, string status);
        Task DeleteOrderAsync(int id);
    }
}

// YourSolution.Services/Implementations/OrderService.cs
namespace YourSolution.Services.Implementations
{
    public class OrderService : IOrderService
    {
        private readonly IOrderRepository _orderRepository;
        private readonly IProductRepository _productRepository;
        private readonly ILogger<OrderService> _logger;

        public OrderService(
            IOrderRepository orderRepository,
            IProductRepository productRepository,
            ILogger<OrderService> logger)
        {
            _orderRepository = orderRepository;
            _productRepository = productRepository;
            _logger = logger;
        }

        public async Task<OrderDto> GetOrderByIdAsync(int id)
        {
            var order = await _orderRepository.GetByIdAsync(id);
            if (order == null)
                throw new NotFoundException($"Order with ID {id} not found");

            return MapToDto(order);
        }

        public async Task<OrderDto> GetOrderByNumberAsync(string orderNumber)
        {
            var order = await _orderRepository.GetByOrderNumberAsync(orderNumber);
            if (order == null)
                throw new NotFoundException($"Order with number {orderNumber} not found");

            return MapToDto(order);
        }

        public async Task<IEnumerable<OrderDto>> GetAllOrdersAsync()
        {
            var orders = await _orderRepository.GetAllAsync();
            return orders.Select(MapToDto);
        }

        public async Task<IEnumerable<OrderDto>> GetOrdersByCustomerIdAsync(string customerId)
        {
            var orders = await _orderRepository.GetByCustomerIdAsync(customerId);
            return orders.Select(MapToDto);
        }

        public async Task<OrderDto> CreateOrderAsync(CreateOrderDto createOrderDto)
        {
            // Validate input
            if (createOrderDto.Items == null || !createOrderDto.Items.Any())
                throw new BadRequestException("Order must contain at least one item");

            // Create new order
            var order = new Order
            {
                OrderNumber = GenerateOrderNumber(),
                OrderDate = DateTime.UtcNow,
                CustomerId = createOrderDto.CustomerId,
                Status = OrderStatus.Pending,
                Items = new List<OrderItem>()
            };

            // Get product details and calculate totals
            decimal totalAmount = 0;

            foreach (var itemDto in createOrderDto.Items)
            {
                var product = await _productRepository.GetByIdAsync(itemDto.ProductId);
                if (product == null)
```

```
                    throw new NotFoundException($"Product with ID {itemDto.ProductId} not found");

                var orderItem = new OrderItem
                {
                    ProductId = product.Id,
                    ProductName = product.Name,
                    UnitPrice = product.Price,
                    Quantity = itemDto.Quantity
                };

                order.Items.Add(orderItem);
                totalAmount += orderItem.UnitPrice * orderItem.Quantity;
            }

            order.TotalAmount = totalAmount;

            // Save to database
            await _orderRepository.CreateAsync(order);
            _logger.LogInformation($"Created new order with number {order.OrderNumber}");

            return MapToDto(order);
        }

        public async Task UpdateOrderStatusAsync(int id, string status)
        {
            var order = await _orderRepository.GetByIdAsync(id);
            if (order == null)
                throw new NotFoundException($"Order with ID {id} not found");

            if (!Enum.TryParse<OrderStatus>(status, true, out var orderStatus))
                throw new BadRequestException($"Invalid order status: {status}");

            order.Status = orderStatus;
            await _orderRepository.UpdateAsync(order);
            _logger.LogInformation($"Updated order {order.OrderNumber} status to {status}");
        }

        public async Task DeleteOrderAsync(int id)
        {
            var order = await _orderRepository.GetByIdAsync(id);
            if (order == null)
                throw new NotFoundException($"Order with ID {id} not found");

            await _orderRepository.DeleteAsync(id);
            _logger.LogInformation($"Deleted order with ID {id}");
        }

        private string GenerateOrderNumber()
        {
            return $"ORD-{DateTime.UtcNow:yyyyMMdd}-{Guid.NewGuid().ToString().Substring(0, 8).ToUpper()}";
        }

        private OrderDto MapToDto(Order order)
        {
            return new OrderDto
            {
                Id = order.Id,
                OrderNumber = order.OrderNumber,
                OrderDate = order.OrderDate,
                TotalAmount = order.TotalAmount,
                CustomerId = order.CustomerId,
                Status = order.Status.ToString(),
                Items = order.Items.Select(item => new OrderItemDto
                {
                    Id = item.Id,
                    ProductId = item.ProductId,
                    ProductName = item.ProductName,
                    UnitPrice = item.UnitPrice,
                    Quantity = item.Quantity
                }).ToList()
            };
        }
    }
}
```

Presentation Layer

Finally, implement the API controller:

csharp

```csharp
// YourSolution.API/DTOs/OrderCreateRequestDto.cs
namespace YourSolution.API.DTOs
{
    public class OrderCreateRequestDto
    {
        [Required]
        public string CustomerId { get; set; }

        [Required]
        [MinLength(1, ErrorMessage = "At least one item is required")]
        public List<OrderItemRequestDto> Items { get; set; } = new List<OrderItemRequestDto>();
    }

    public class OrderItemRequestDto
    {
        [Required]
        public int ProductId { get; set; }

        [Required]
        [Range(1, int.MaxValue, ErrorMessage = "Quantity must be at least 1")]
        public int Quantity { get; set; }
    }
}

// YourSolution.API/Controllers/OrdersController.cs
namespace YourSolution.API.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class OrdersController : ControllerBase
    {
        private readonly IOrderService _orderService;
        private readonly ILogger<OrdersController> _logger;

        public OrdersController(
            IOrderService orderService,
            ILogger<OrdersController> logger)
        {
            _orderService = orderService;
            _logger = logger;
        }

        [HttpGet]
        [ProducesResponseType(StatusCodes.Status200OK)]
        public async Task<ActionResult<IEnumerable<OrderDto>>> GetOrders()
        {
            var orders = await _orderService.GetAllOrdersAsync();
            return Ok(orders);
        }

        [HttpGet("{id:int}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<ActionResult<OrderDto>> GetOrderById(int id)
        {
            try
            {
                var order = await _orderService.GetOrderByIdAsync(id);
                return Ok(order);
            }
            catch (NotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
        }

        [HttpGet("by-number/{orderNumber}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<ActionResult<OrderDto>> GetOrderByNumber(string orderNumber)
        {
            try
            {
                var order = await _orderService.GetOrderByNumberAsync(orderNumber);
                return Ok(order);
            }
            catch (NotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
        }

        [HttpGet("customer/{customerId}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        public async Task<ActionResult<IEnumerable<OrderDto>>> GetOrdersByCustomerId(string customerId)
        {
            var orders = await _orderService.GetOrdersByCustomerIdAsync(customerId);
            return Ok(orders);
        }

        [HttpPost]
        [ProducesResponseType(StatusCodes.Status201Created)]
        [ProducesResponseType(StatusCodes.Status400BadRequest)]
        public async Task<ActionResult<OrderDto>> CreateOrder(OrderCreateRequestDto request)
        {
            try
            {
                var createOrderDto = new CreateOrderDto
                {
                    CustomerId = request.CustomerId,
                    Items = request.Items.Select(i => new CreateOrderItemDto
                    {
                        ProductId = i.ProductId,
                        Quantity = i.Quantity
                    }).ToList()
                };

                var createdOrder = await _orderService.CreateOrderAsync(createOrderDto);

                return CreatedAtAction(
                    nameof(GetOrderById),
                    new { id = createdOrder.Id },
                    createdOrder);
            }
            catch (BadRequestException ex)
            {
                return BadRequest(new { message = ex.Message });
            }
            catch (NotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
        }

        [HttpPut("{id:int}/status")]
        [ProducesResponseType(StatusCodes.Status204NoContent)]
        [ProducesResponseType(StatusCodes.Status400BadRequest)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<IActionResult> UpdateOrderStatus(int id, [FromBody] string status)
        {
            try
            {
                await _orderService.UpdateOrderStatusAsync(id, status);
                return NoContent();
            }
            catch (NotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
            catch (BadRequestException ex)
            {
                return BadRequest(new { message = ex.Message });
            }
        }

        [HttpDelete("{id:int}")]
        [ProducesResponseType(StatusCodes.Status204NoContent)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        public async Task<IActionResult> DeleteOrder(int id)
        {
            try
            {
                await _orderService.DeleteOrderAsync(id);
                return NoContent();
            }
            catch (NotFoundException ex)
            {
                return NotFound(new { message = ex.Message });
            }
        }
    }
}
```

Dependency Injection Setup

Configure the dependency injection in the `Program.cs` file:

csharp

```csharp
// YourSolution.API/Program.cs
var builder = WebApplication.CreateBuilder(args);

// Add DbContext
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Register repositories
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IProductRepository, ProductRepository>();

// Register services
builder.Services.AddScoped<IOrderService, OrderService>();

// Add controllers
builder.Services.AddControllers();

// Add API documentation
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure middleware
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

5.4 Benefits of Layered Architecture

1. Separation of Concerns

Each layer has a specific responsibility, making the codebase easier to understand. Developers can focus on one aspect of the application at a time without having to understand the entire system.

2. Maintainability

Changes to one layer typically don't affect other layers as long as the interfaces remain consistent. This makes it easier to update or fix issues in isolation.

3. Testability

Layers can be tested in isolation using mocks or stubs for dependencies, enabling more thorough unit testing. For example, service tests don't need a real database, just a mocked repository implementation.

4. Flexibility

Implementations can be swapped without affecting other parts of the application. For instance, you could change the database from SQL Server to PostgreSQL by only modifying the repository layer.

5. Reusability

Components within layers can often be reused across different parts of the application or even in different applications. Common services or repositories might be shared among multiple controllers.

6. Organization and Scalability

The clear structure helps teams collaborate more effectively, as responsibilities are clearly defined. New team members can more easily understand the architecture and start contributing.

7. Security

Business logic is protected from direct external access, as it's only accessible through the API layer. This prevents bypassing important validation or business rules.

5.5 Disadvantages of Layered Architecture

1. Overhead for Simple Applications

For small applications, the extra layers can add unnecessary complexity. Simple CRUD operations need to pass through multiple layers when they could be handled directly.

2. Performance Impact

Data often needs to be transformed as it moves between layers (entity to DTO to response model), which can impact performance for high-throughput applications.

3. Boilerplate Code

There's often repetitive code across layers, particularly for mapping between different object types. For example:

- Entity to DTO mapping in services
- DTO to view model mapping in controllers

4. More Files and Interfaces

The number of files and interfaces can grow quickly, potentially making navigation more difficult. Each entity might have:

- Domain model
- Repository interface
- Repository implementation
- Service interface
- Service implementation
- Multiple DTOs
- Controller

5. Tight Coupling to Architectural Pattern

Applications can become overly committed to the layered pattern, making architectural changes difficult. Future requirements might call for a different architectural style that's hard to adopt.

6. "Pass-through" Methods

Services sometimes just delegate to repositories without adding value, creating unnecessary indirection. For example:

csharp

```csharp
// Service that doesn't add any value
public async Task<OrderDto> GetOrderByIdAsync(int id)
{
    var order = await _orderRepository.GetByIdAsync(id);
    return MapToDto(order); // Only does simple mapping
}
```

7. Potential for Dependency Cycles

Without careful design, layers can develop circular dependencies, especially when lower layers need to call higher layers for certain operations.

5.6 Best Practices for Layered Architecture

1. **Keep layers loosely coupled**: Use interfaces and dependency injection to minimize direct dependencies between implementation classes.
2. **Follow the Dependency Rule**: Dependencies should point inward. Outer layers can depend on inner layers, but inner layers should not depend on outer layers.
3. **Use DTOs at boundaries**: Create Data Transfer Objects to pass data between layers instead of passing domain entities directly, especially when crossing application boundaries.
4. **Consider using AutoMapper**: Tools like AutoMapper can reduce the boilerplate mapping code between different object types.

5. **Don't force layering when unnecessary**: If a service is just passing through to a repository without adding value, consider if that layer is truly needed for that specific operation.
6. **Use exception handling strategically**: Catch specific exceptions at appropriate layers. For example:
    - Repository layer: Handle data access exceptions
    - Service layer: Handle business rule violations
    - Controller layer: Transform exceptions into HTTP responses
7. **Add value at each layer**: Each layer should have a clear purpose and add value:
    - Controllers: Handle HTTP concerns, validation, authentication
    - Services: Handle business logic, orchestration, transactions
    - Repositories: Handle data access and persistence
8. **Consider vertical slicing for complex applications**: For large applications, consider organizing by feature rather than just by layer, combining the benefits of layered architecture with domain-driven design.

5.7 Alternatives to Traditional Layered Architecture

While traditional layered architecture works well for many applications, there are alternatives that might be better suited for specific scenarios:

1. **Vertical Slice Architecture**: Organizes code by feature rather than by technical function, reducing the need to jump between layers.
2. **CQRS (Command Query Responsibility Segregation)**: Separates read and write operations, allowing them to be optimized independently.
3. **Clean Architecture / Onion Architecture**: Places the domain model at the center, with all dependencies pointing inward.
4. **Microservices Architecture**: Divides the application into independent services, each with its own internal layering.
5. **Serverless Architecture**: Focuses on individual functions rather than layers, leveraging cloud provider capabilities.

Choosing the right architecture depends on your specific requirements, team expertise, and project constraints. Layered architecture provides a solid foundation that can evolve as your application grows.