ASP.NET Core Web API

Core Concepts

1. MVC (Model-View-Controller) Architecture

MVC is an architectural pattern that separates an application into three main components:

- **Model**: Represents the data structure and business logic. For an e-shop, models would represent entities like `Product`, `Order`, `Customer`, etc.
- **View**: Defines the user interface. In an API context, the view layer is handled by the client (like Angular, React) that consumes the API.
- **Controller**: Acts as an intermediary between the Model and View. It handles HTTP requests, processes input, and returns data.

2. In ASP.NET Core, **Controllers** are the central part of the MVC architecture and are responsible for handling incoming HTTP requests, processing them, and returning appropriate responses. Controllers allow you to define routes, perform actions based on the request data, and return the results in formats like JSON, XML, or HTML.

Here's an in-depth look at controllers, how to create them, and their features, with examples to illustrate their usage.

1. What is a Controller?

A controller in ASP.NET Core is a class that inherits from `ControllerBase` or `Controller` (if you're using MVC with views). It serves as the entry point for client requests and contains action methods that correspond to HTTP operations (GET, POST, PUT, DELETE, etc.).

Each action method in the controller is mapped to a specific endpoint and is responsible for executing business logic and returning responses to the client.

Example of a Basic Controller:

```
using Microsoft.AspNetCore.Mvc;

namespace EShop.Api.Controllers
{       [Route("api/[controller]")]
        [ApiController]
        public class ProductsController : ControllerBase
        {       [HttpGet]
        public IActionResult GetProducts()
        {       // Retrieve products from a service or database
        return Ok(products); // Returns 200 OK response with data       }
        [HttpGet("{id}")]
        public IActionResult GetProductById(int id)
        {       // Retrieve a single product by ID
        var product = productService.GetProductById(id);
        if (product == null)
        return NotFound(); // Returns 404 if not found
        return Ok(product); // Returns 200 OK with product data
        }}}
```

2. Defining a Controller

Controllers should:

1. **Inherit from `ControllerBase`** if they're intended for Web APIs (without views).
2. **Inherit from `Controller`** if using MVC views (HTML, Razor Pages, etc.).

By default, in a Web API project, controllers use `ControllerBase` for a leaner API experience without view-related functionality.

Example of ControllerBase vs. Controller:

```
// Web API Controller

public class ProductsController : ControllerBase

{

        // API-specific logic

}

// MVC Controller (used with views)

public class HomeController : Controller

{

        // MVC-specific logic, returning views

}
```

Routing in Controllers

Routing is how HTTP requests are mapped to action methods in controllers. In ASP.NET Core, there are two primary types of routing:

- **Attribute Routing**: Defines routes with attributes directly on controllers and action methods.
- **Convention-based Routing**: Configures routes globally in `Program.cs`.

Attribute Routing

In API development, **attribute routing** is more commonly used as it allows fine-grained control over routes at the controller and action levels.

Example of Attribute Routing:

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
        // Matches GET /api/products
        [HttpGet]
        public IActionResult GetAllProducts() => Ok(_productService.GetAll());
        // Matches GET /api/products/{id}
        [HttpGet("{id}")]
        public IActionResult GetProductById(int id) => Ok(_productService.GetById(id));
        // Matches POST /api/products
        [HttpPost]
        public IActionResult AddProduct(ProductDto productDto) => Ok(_productService.Add(productDto));
}
```

Route Tokens in Attribute Routing:

- `[controller]`: Replaced with the controller's name (without the "Controller" suffix).
- `[action]`: Replaced with the action method name.

For example, if the controller is `ProductsController`, `[controller]` becomes "products."

4. Action Methods

Action methods are the core methods in controllers that handle specific HTTP requests. They are marked with HTTP verbs (`[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]`) to define the type of HTTP operation they handle.

Example of Action Methods:

```
[HttpGet]  // Handles GET requests
public IActionResult GetAllProducts() { /* logic */ }

[HttpPost] // Handles POST requests
public IActionResult CreateProduct(ProductDto productDto) { /* logic */ }

[HttpPut("{id}")] // Handles PUT requests with an ID in the URL
public IActionResult UpdateProduct(int id, ProductDto productDto) { /* logic */ }

[HttpDelete("{id}")] // Handles DELETE requests
public IActionResult DeleteProduct(int id) { /* logic */ }
```

Returning Responses

ASP.NET Core provides several helper methods for standard HTTP responses:

- **Ok(object)**: Returns `200 OK` with data in the response body.
- **NotFound()**: Returns `404 Not Found` if the requested resource does not exist.
- **CreatedAtAction()**: Returns `201 Created` for newly created resources, with a link to the new resource.
- **BadRequest()**: Returns `400 Bad Request` if the client request is invalid.
- **NoContent()**: Returns `204 No Content` for successful operations with no content in the response.

Example of Different Response Types:

```
public IActionResult CreateProduct(ProductDto productDto)
{
        if (!ModelState.IsValid)
        return BadRequest(ModelState); // 400 Bad Request

        var product = _productService.Add(productDto);
        return CreatedAtAction(nameof(GetProductById), new { id = product.Id }, product);
// 201 Created
}
```

`IActionResult` is an interface in ASP.NET Core that represents the result of an action method in a controller. It provides a flexible way for a controller action to return different types of responses, such as `200 OK`, `404 Not Found`, or `500 Internal Server Error`, without being tied to a specific type of response.

When an action method returns `IActionResult`, it allows the method to:

- Return different HTTP status codes based on the outcome.
- Include data, status codes, and headers in the response.
- Allow the framework to handle formatting the response correctly (e.g., JSON or XML).

`IActionResult` is commonly used in Web API projects to make controller actions more flexible and expressive.

Dependency Injection in Controllers

ASP.NET Core allows injecting services into controllers through the constructor. This follows Dependency Injection (DI) principles, which makes the code cleaner, easier to test, and more maintainable.

Example of Dependency Injection:

```
public class ProductsController : ControllerBase
{       private readonly IProductService _productService;
        public ProductsController(IProductService productService)
        {       _productService = productService;       }
        [HttpGet]
        public IActionResult GetAllProducts()
        {
        var products = _productService.GetAllProducts();
        return Ok(products);
        }
}
```

Model Binding and Validation

ASP.NET Core automatically binds data from HTTP requests (such as JSON data) to parameters in action methods.

For example, in a `POST` request, data can be bound to a model:

```
[HttpPost]
public IActionResult AddProduct([FromBody] ProductDto productDto)
{
        if (!ModelState.IsValid)
        return BadRequest(ModelState); // 400 if validation fails

        var createdProduct = _productService.AddProduct(productDto);
        return CreatedAtAction(nameof(GetProductById), new { id = createdProduct.Id }, createdProduct);
}
```

Model Binding: Binds the JSON request body to the `ProductDto` parameter.

Validation: `ModelState.IsValid` checks if the model meets data annotations or custom validation rules.

Asynchronous Controllers

For better scalability, controllers can handle requests asynchronously using the `async` and `await` keywords.

Example of an Asynchronous Controller Action:

```
[HttpGet("{id}")]
public async Task<IActionResult> GetProductById(int id)
{
    var product = await _productService.GetProductByIdAsync(id);
    return product != null ? Ok(product) : NotFound();
}
```

Filters in Controllers

Filters in ASP.NET Core allow you to add reusable logic before or after controller actions are executed.

Common filters include:

- **Authorization Filters**: Check if the user is authorized.
- **Action Filters**: Run code before or after an action method executes.
- **Exception Filters**: Handle exceptions thrown by action methods.
- **Result Filters**: Run code before or after the action result is returned.

Example of Applying Filters:

```
[Authorize] // Authorization filter
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
{       [HttpGet]
        [ServiceFilter(typeof(LoggingActionFilter))] // Custom action filter
        public IActionResult GetOrders()
        {       // ...    }
}
```

Controller Testing

Controllers in ASP.NET Core can be tested with unit tests, where you mock dependencies (such as services) and check that controllers return the expected HTTP responses.

Example of a Simple Unit Test for Controller:

```
public class ProductsControllerTests

{

        private readonly ProductsController _controller;

        private readonly Mock<IProductService> _mockProductService;

        public ProductsControllerTests()

        {

        _mockProductService = new Mock<IProductService>();

        _controller = new ProductsController(_mockProductService.Object);

        }

        [Fact]

        public void GetAllProducts_ReturnsOkResult()

        {

        var result = _controller.GetAllProducts();

        Assert.IsType<OkObjectResult>(result);

        }

}
```

In Entity Framework Core, the `DbContext` class represents a session with the database, allowing you to query and save data. It acts as a bridge between your application and the database, enabling you to map database tables to .NET classes (entities), manage transactions, and enforce rules on data interaction.

DbContext Definition and Usage

The `DbContext` class in EF Core is a representation of the database session. Typically, you create a class that inherits from `DbContext` and contains properties of type `DbSet<T>` for each entity that you want to work with.

Example of Defining a Custom DbContext:

```
using Microsoft.EntityFrameworkCore;
using EShop.Domain.Entities;
namespace EShop.Infrastructure.Data
{       public class EShopDbContext : DbContext
        {
        public EShopDbContext(DbContextOptions<EShopDbContext> options)
        : base(options)
        {
        }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Payment> Payments { get; set; }
        }
}
```

In this example:

- `EShopDbContext` inherits from `DbContext`.
- `DbSet<T>` properties (`Products`, `Orders`, etc.) represent tables in the database.

Each `DbSet` property corresponds to a table in the database, where each row in the table maps to an entity.

2. DbSets and Entity Mapping

`DbSet<T>` represents a table in the database and provides methods for querying and saving instances of `T` (the entity class).

- **DbSet<T>**: A property of type `DbSet<T>` in the context maps to a table where each instance of `T` represents a row.
- **CRUD Operations**: `DbSet<T>` provides methods like `Add`, `Remove`, `Find`, and `Update` for performing CRUD operations.

Example of a DbSet Property for Product Entity:

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

This maps the `Products` property to a `Product` table in the database, enabling you to perform operations like:

```
var product = new Product { Name = "Laptop", Price = 1000 };

context.Products.Add(product);

context.SaveChanges();
```

Configuring DbContext with Dependency Injection

In ASP.NET Core applications, `DbContext` is typically registered in `Program.cs` to use dependency injection (DI).

Example of Configuring DbContext in Program.cs:

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

builder.Services.AddDbContext<EShopDbContext>(options =>

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

var app = builder.Build();
```

This configuration:

- Registers `EShopDbContext` with DI, making it available for injection in other classes (e.g., repositories, services).
- Specifies that EF Core should use SQL Server with a connection string from the configuration file.

4. Configuring Connection Strings

The connection string tells EF Core where to find the database. In ASP.NET Core, connection strings are usually stored in `appsettings.json`.

Example `appsettings.json`:

```
{
  "ConnectionStrings": {
        "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=EShopDb;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

In `Program.cs`, retrieve the connection string using the configuration object.

OnModelCreating and Fluent API

The `OnModelCreating` method allows you to configure entity properties, relationships, and constraints using the Fluent API. This provides more control over the mapping between .NET classes and database tables.

Example of Using `OnModelCreating` for Entity Configuration:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
        modelBuilder.Entity<Product>()
        .ToTable("Products")
        .HasKey(p => p.Id);
        modelBuilder.Entity<Product>()
        .Property(p => p.Name)
        .IsRequired()
        .HasMaxLength(100);
        modelBuilder.Entity<Order>()
        .HasMany(o => o.OrderItems)
        .WithOne()
        .HasForeignKey(oi => oi.OrderId);
}
```

In this example:

- The `Products` table is mapped to the `Product` entity with `Id` as the primary key.
- `Name` is set as a required property with a max length of 100.
- An `Order` entity has a one-to-many relationship with `OrderItems`.

6. DbContext Options

When configuring `DbContext`, you can pass options such as:

- **Lazy Loading**: Loads related entities only when they’re accessed.
- **Logging**: Logs SQL queries for debugging.

Example of Enabling SQL Logging:

options.UseSqlServer(connectionString)

.EnableSensitiveDataLogging(); // Caution: Only for development

Transactions in DbContext

EF Core provides transaction management through `SaveChanges()` and `BeginTransaction()`. Transactions ensure that a group of operations either all succeed or all fail.

Example of Using Transactions:

```
using (var transaction = await context.Database.BeginTransactionAsync())
{
        try
        {       // Execute multiple operations
        context.Products.Add(new Product { Name = "Tablet", Price = 500 });
        context.SaveChanges();
        context.Customers.Add(new Customer { Name = "John Doe" });
        context.SaveChanges();
        transaction.Commit();
        }
        catch
        {
        transaction.Rollback();
        }
}
```

Tracking and No-Tracking Queries

By default, EF Core **tracks** entities in the `DbContext`, so changes are automatically detected and saved. However, for read-only operations, **No-Tracking** queries are more efficient.

- **Tracked Queries**: Used when you need to modify data.
- **No-Tracking Queries**: Used for read-only operations, improving performance.

Example of No-Tracking Query:

var products = context.Products.AsNoTracking().ToList();

Managing DbContext Lifetime

In ASP.NET Core, the lifetime of `DbContext` is managed through DI. The default and recommended lifetime for `DbContext` is **Scoped**, which means each request gets its own instance of `DbContext`.

DbContext Lifetimes:

- **Scoped** (default): One `DbContext` per request.
- **Transient**: New instance per dependency injection.
- **Singleton**: Shared `DbContext` across the application (not recommended for DbContext).

10. Seeding Data

Data seeding populates the database with initial data. In EF Core, you can seed data in the `OnModelCreating` method.

Example of Data Seeding in `OnModelCreating`:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)

{

        modelBuilder.Entity<Product>().HasData(

        new Product { Id = 1, Name = "Laptop", Price = 1200 },

        new Product { Id = 2, Name = "Smartphone", Price = 800 }

        );

}
```

The `HasData` method adds records that EF Core will insert into the database the first time it is created or during migrations.

Summary

`DbContext` is a powerful component in Entity Framework Core that simplifies database access and management. Key features include:

- **DbSet Properties**: Define tables in the database.
- **Dependency Injection**: Allows for flexible and scalable application design.
- **OnModelCreating Method**: Customize entity configuration, relationships, and constraints.
- **Transactions**: Ensures data consistency in complex operations.
- **Tracking and No-Tracking Queries**: Optimizes performance by choosing appropriate tracking.
- **Data Seeding**: Allows for initial data population.

A well-configured `DbContext` helps you handle data interactions, maintain consistency, and follow best practices in building efficient and reliable applications.

Dependency Injection (DI) and Services Layer in ASP.NET Core

Dependency Injection (DI) is a design pattern that promotes loose coupling and enhances testability and maintainability in applications. In ASP.NET Core, DI is a fundamental part of the framework, providing built-in support for injecting dependencies into classes, such as controllers and services.

The **Services Layer**, on the other hand, encapsulates the business logic of the application. It acts as an intermediary between controllers (which handle HTTP requests) and repositories or data access layers (which interact with the database). The Services Layer processes data, enforces business rules, and ensures that the application behaves correctly.

This detailed explanation covers:

1. **Understanding Dependency Injection (DI)**
    - What is Dependency Injection?
    - Benefits of DI
    - How DI works in ASP.NET Core
    - Registering services in the DI container
    - Injecting dependencies into classes
2. **The Services Layer**
    - Role of the Services Layer
    - Designing the Services Layer
    - Interaction with other layers
    - Best practices
3. **Implementing DI and Services Layer in an E-Shop Application**
    - Example code for DI
    - Example code for Services Layer
    - Putting it all together

Understanding Dependency Injection (DI)

What is Dependency Injection?

Dependency Injection is a design pattern where an object (the **dependency**) is passed to a class, rather than the class creating the object itself. This allows for greater decoupling between classes, making the code more modular, testable, and easier to maintain.

In traditional programming, classes create their own dependencies via the `new` keyword. This tight coupling makes it difficult to test classes in isolation and to change dependencies without modifying the classes themselves.

Dependency Injection Principles:

- **Inversion of Control (IoC):** The control of object creation is inverted. Instead of a class instantiating its dependencies, they are provided from an external source.
- **Loose Coupling:** Classes depend on abstractions (interfaces), not on concrete implementations.
- **Service Lifetimes:** Control over the lifetime of services (e.g., singleton, scoped, transient).

Benefits of DI

- **Improved Testability:** Dependencies can be easily mocked or stubbed during unit testing.
- **Loose Coupling:** Changes to one class have minimal impact on others.
- **Flexibility and Extensibility:** Easy to swap implementations without changing dependent classes.
- **Maintainability:** Clear separation of concerns enhances code readability and maintainability.

How DI Works in ASP.NET Core

ASP.NET Core has a built-in **Dependency Injection container** (also known as the **Service Container**). It manages the registration and resolution of services.

Key aspects:

- **Service Registration:** Register services with the DI container in `Program.cs` or `Startup.cs` (depending on the ASP.NET Core version).
- **Service Resolution:** The framework injects the required services into constructors or methods.
- **Service Lifetimes:** Specify how long services should live (Singleton, Scoped, Transient).

Registering Services in the DI Container

Services are registered in the DI container during application startup. In ASP.NET Core 6 and later, this is typically done in `Program.cs`.

Service Lifetimes:

- **Singleton:** One instance for the entire application lifetime.
- **Scoped:** One instance per HTTP request.
- **Transient:** A new instance each time the service is requested.

Example of Service Registration:

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

// Register services
builder.Services.AddSingleton<ISingletonService, SingletonService>();
builder.Services.AddScoped<IScopedService, ScopedService>();
builder.Services.AddTransient<ITransientService, TransientService>();

// Register DbContext with Scoped lifetime (default)
builder.Services.AddDbContext<EShopDbContext>(options =>
        options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
```

Injecting Dependencies into Classes

Dependencies are typically injected via constructors. ASP.NET Core automatically resolves the dependencies from the DI container.

Example of Constructor Injection in a Controller:

```
public class ProductsController : ControllerBase
{
        private readonly IProductService _productService;
        // Dependency Injection via constructor
        public ProductsController(IProductService productService)
        {
        _productService = productService;
        }
        // Action methods using _productService
}
```

Injecting into Services:

Services can have dependencies on other services or repositories, which can also be injected via constructors.

```
public class ProductService : IProductService
{
        private readonly IProductRepository _productRepository;

        // Dependency Injection
        public ProductService(IProductRepository productRepository)
        {
        _productRepository = productRepository;
        }

        // Business logic methods
}
```

Field Injection and Property Injection: While possible, these methods are generally discouraged in favor of constructor injection for better testability and clarity.

The Services Layer

Role of the Services Layer

The Services Layer encapsulates the business logic of the application. It acts as a mediator between the controllers (presentation layer) and the data access layer (repositories or data context). The main responsibilities include:

- **Implementing Business Logic:** Contains rules and operations that are specific to the business domain.
- **Data Transformation:** Transforms data between different layers, such as mapping entities to DTOs (Data Transfer Objects).
- **Orchestrating Operations:** Coordinates complex operations that may involve multiple repositories or external services.
- **Validation:** Validates data according to business rules before processing or persisting it.

Designing the Services Layer

- **Interfaces:** Define interfaces (e.g., `IProductService`) to abstract the implementation and promote loose coupling.
- **Implementation Classes:** Concrete classes (e.g., `ProductService`) implement these interfaces.
- **Separation of Concerns:** Keep business logic separate from data access logic.

Interaction with Other Layers

- **Controllers:** Use services to handle HTTP requests and responses.
- **Repositories/Data Access Layer:** Services interact with repositories to retrieve and persist data.
- **Models/Entities:** Services may map entities to DTOs or vice versa.

Flow Example:

1. **Controller** receives an HTTP request and calls a method on a service.
2. **Service** processes the request, applies business logic, and interacts with repositories as needed.
3. **Repository** handles data access operations with the database via `DbContext`.
4. **Service** returns the result to the controller.
5. **Controller** returns the response to the client.

Best Practices

- **Use Interfaces:** Define interfaces for services to enable loose coupling and easier testing.
- **Single Responsibility Principle:** Each service should have a clear purpose.
- **Avoid Fat Controllers:** Controllers should be thin and delegate business logic to services.
- **Exception Handling:** Services should handle exceptions and return appropriate error information.
- **Validation:** Implement validation logic within services or use separate validation layers.

Implementing DI and Services Layer in an E-Shop Application

Let's apply these concepts in the context of an e-commerce application (e-shop).

Example Code for Dependency Injection

Registering Services in Program.cs:

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

// Register DbContext

builder.Services.AddDbContext<EShopDbContext>(options =>

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

// Register Repositories

builder.Services.AddScoped<IProductRepository, ProductRepository>();

builder.Services.AddScoped<IOrderRepository, OrderRepository>();

builder.Services.AddScoped<ICustomerRepository, CustomerRepository>();

// Register Services

builder.Services.AddScoped<IProductService, ProductService>();

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

builder.Services.AddScoped<ICustomerService, CustomerService>();

var app = builder.Build();
```

In this code:

- **DbContext** is registered as Scoped (default).
- **Repositories and Services** are registered as Scoped, meaning a new instance per HTTP request.
- **Interfaces and Implementations:** Each service and repository has an interface and a concrete implementation.

Example Code for Services Layer

Service Interface - `IProductService.cs`:

```
using EShop.Application.DTOs;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace EShop.Application.Interfaces

{		public interface IProductService

		{

		Task<IEnumerable<ProductDto>> GetAllProductsAsync();

		Task<ProductDto> GetProductByIdAsync(int id);

		Task AddProductAsync(ProductDto productDto);

		Task<bool> UpdateProductAsync(int id, ProductDto productDto);

		Task<bool> DeleteProductAsync(int id);

		}

}
```

Service Implementation - `ProductService.cs`:

```
using EShop.Application.DTOs;

using EShop.Application.Interfaces;

using EShop.Domain.Entities;

using EShop.Infrastructure.Repositories;

using System.Collections.Generic;

using System.Threading.Tasks;

using AutoMapper;

namespace EShop.Application.Services

{               public class ProductService : IProductService

                {

                private readonly IProductRepository _productRepository;

                private readonly IMapper _mapper;

                // Inject dependencies

                public ProductService(IProductRepository productRepository, IMapper mapper)

                {                _productRepository = productRepository;

                                _mapper = mapper;

                }
```

```
public async Task<IEnumerable<ProductDto>> GetAllProductsAsync()
{
var products = await _productRepository.GetAllAsync();
return _mapper.Map<IEnumerable<ProductDto>>(products);
}
public async Task<ProductDto> GetProductByIdAsync(int id)
{
var product = await _productRepository.GetByIdAsync(id);
return product != null ? _mapper.Map<ProductDto>(product) : null;
}
public async Task AddProductAsync(ProductDto productDto)
{
var product = _mapper.Map<Product>(productDto);
await _productRepository.AddAsync(product);
}
```

```
public async Task<bool> UpdateProductAsync(int id, ProductDto productDto)

{var existingProduct = await _productRepository.GetByIdAsync(id);

if (existingProduct == null)

return false;

// Update properties

existingProduct.Name = productDto.Name;

existingProduct.Description = productDto.Description;

existingProduct.Price = productDto.Price;

existingProduct.Stock = productDto.Stock;

await _productRepository.UpdateAsync(existingProduct);

return true;

}

public async Task<bool> DeleteProductAsync(int id)

{                    var product = await _productRepository.GetByIdAsync(id);

if (product == null)

return false;

await _productRepository.DeleteAsync(product);

return true;

}}}
```

In this service:

- **Dependencies:** `IProductRepository` and `IMapper` are injected via constructor injection.
- **Business Logic:** The service methods perform operations like data mapping and interact with the repository.
- **Data Transformation:** Uses AutoMapper to map between `Product` entities and `ProductDto`.

Repository Interface - `IProductRepository.cs`:

```
using EShop.Domain.Entities;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace EShop.Infrastructure.Repositories
{
        public interface IProductRepository
        {
        Task<IEnumerable<Product>> GetAllAsync();
        Task<Product> GetByIdAsync(int id);
        Task AddAsync(Product product);
        Task UpdateAsync(Product product);
        Task DeleteAsync(Product product);
        }
}
```

Repository Implementation - `ProductRepository.cs`:

```
using EShop.Domain.Entities;
using EShop.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace EShop.Infrastructure.Repositories
{ public class ProductRepository : IProductRepository
{
private readonly EShopDbContext _context;
public ProductRepository(EShopDbContext context)
{
_context = context;
}
public async Task<IEnumerable<Product>> GetAllAsync()
{
return await _context.Products.AsNoTracking().ToListAsync();
}
```

```
public async Task<Product> GetByIdAsync(int id)
{           return await _context.Products.FindAsync(id);      }
public async Task AddAsync(Product product)
{           await _context.Products.AddAsync(product);
await _context.SaveChangesAsync();
}
public async Task UpdateAsync(Product product)
{  _context.Products.Update(product);
await _context.SaveChangesAsync();
}
public async Task DeleteAsync(Product product)
{           _context.Products.Remove(product);
await _context.SaveChangesAsync();
}}}
```

Putting It All Together

Controller - `ProductsController.cs`:

```
using Microsoft.AspNetCore.Mvc;
using EShop.Application.Interfaces;
using EShop.Application.DTOs;
using System.Threading.Tasks;
namespace EShop.Api.Controllers
{
        [Route("api/[controller]")]
        [ApiController]
        public class ProductsController : ControllerBase
        {
        private readonly IProductService _productService;
        // Dependency Injection
        public ProductsController(IProductService productService)
        {
        _productService = productService;
        }
```

```
// GET: api/products

[HttpGet]

public async Task<IActionResult> GetAllProducts()

{                    var products = await _productService.GetAllProductsAsync();

return Ok(products);            }

// GET: api/products/{id}

[HttpGet("{id}")]

public async Task<IActionResult> GetProductById(int id)

{                    var product = await _productService.GetProductByIdAsync(id);

return product != null ? Ok(product) : NotFound();          }

// POST: api/products

[HttpPost]

public async Task<IActionResult> AddProduct([FromBody] ProductDto productDto)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

await _productService.AddProductAsync(productDto);

return CreatedAtAction(nameof(GetProductById), new { id = productDto.Id }, productDto);

}
```

```
        // PUT: api/products/{id}
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateProduct(int id, [FromBody] ProductDto productDto)
        {
        if (!ModelState.IsValid)
        return BadRequest(ModelState);
        var result = await _productService.UpdateProductAsync(id, productDto);
        return result ? NoContent() : NotFound();
        }
        // DELETE: api/products/{id}
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteProduct(int id)
        {
        var result = await _productService.DeleteProductAsync(id);
        return result ? NoContent() : NotFound();
        }
        }
}
```

Explanation:

- **Controller Layer:** `ProductsController` handles HTTP requests and uses `IProductService`.
- **Services Layer:** `ProductService` contains business logic and interacts with `IProductRepository`.
- **Repositories Layer:** `ProductRepository` handles data access with the database via `EShopDbContext`.
- **Dependency Injection:** All dependencies are injected via constructors, promoting loose coupling.

AutoMapper Configuration:

AutoMapper is used for mapping between entities and DTOs.

Mapping Profile - `MappingProfile.cs`:

```
using AutoMapper;
using EShop.Domain.Entities;
using EShop.Application.DTOs;
namespace EShop.Application.Mapping
{
        public class MappingProfile : Profile
        {
        public MappingProfile()
        {
        CreateMap<Product, ProductDto>().ReverseMap();
        // Other mappings
        }
        }
}
```

Registering AutoMapper in Program.cs:

builder.Services.AddAutoMapper(typeof(MappingProfile));

Summary

- **Dependency Injection (DI):**
  - **Purpose:** Promote loose coupling, improve testability, and manage object lifetimes.
  - **Implementation in ASP.NET Core:** Use the built-in DI container to register and resolve services.
  - **Best Practices:** Use constructor injection, register interfaces with their concrete implementations, and choose appropriate service lifetimes.
- **Services Layer:**
  - **Role:** Encapsulate business logic, validate data, and coordinate operations between controllers and data access layers.
  - **Design:** Define interfaces for services, implement business logic in service classes, and keep controllers thin.
  - **Interaction:** Services interact with repositories and other services, transforming data as needed.
- **In an E-Shop Application:**
  - **Controllers** handle HTTP requests and responses.
  - **Services** process data and enforce business rules.
  - **Repositories** perform data access operations.
  - **Entities/Models** represent the data structure.
  - **DTOs** are used for data transfer between layers.
  - **Dependency Injection** is used throughout to inject services and repositories.