.NET Microservices – Azure DevOps and AKS

Section 3: Products Microservice - Notes

Entities in DataAccessLayer

The Product class represents the entity model for products in the Products microservice. It defines the structure of the Product data that will be stored in the database.

Concepts

1. Entity Framework Core (EF Core) Entity

- **Entity**:
  - Represents a table in the database. Each instance of the entity corresponds to a row in the table.
- **DataAnnotations**:
  - Used for configuring entity properties and validation in EF Core.
- **Primary Key**:
  - Defined by [Key], it uniquely identifies each record in the table.

Code Walkthrough

Product Class

```
using System.ComponentModel.DataAnnotations;


namespace eCommerce.DataAccessLayer.Entities;


public class Product
{
  [Key]
  public Guid ProductID { get; set; }


  public string ProductName { get; set; }
  public string Category { get; set; }
  public double? UnitPrice { get; set; }

  public int? QuantityInStock { get; set; }

}
```

- **ProductID**:
  - **Type**: Guid
  - **Attributes**: [Key]
  - **Description**: Unique identifier for each product. Marked with [Key] to indicate that it is the primary key of the Product table.
- **ProductName**:
  - **Type**: string
  - **Description**: Name of the product.
- **Category**:
  - **Type**: string
  - **Description**: Category under which the product is listed.
- **UnitPrice**:
  - **Type**: double?
  - **Description**: Price of a single unit of the product. It is nullable (double?) because it might not always be specified.
- **QuantityInStock**:
  - **Type**: int?
  - **Description**: Number of units of the product available in stock. It is nullable (int?) because it might not always be specified.

Summary of Key Concepts of "Entities"

1. **Entity Definition**:
   - Product class defines the structure of the Product table in the database, including properties and their types.
2. **Primary Key**:
   - ProductID is marked with [Key] to indicate it is the primary key for the Product table.
3. **Nullable Properties**:
   - UnitPrice and QuantityInStock are nullable to handle cases where these values might not be provided.

The Product class is a straightforward representation of a product in the system, with attributes that define its unique identifier, name, category, price, and stock quantity.

DbContext in DataAccessLayer

The ApplicationDbContext class is a key component of the data access layer in the Products microservice, responsible for interacting with the database using Entity Framework Core (EF Core).

Concepts

1. DbContext

- **DbContext**:
    - Represents a session with the database and provides methods for querying and saving data. It acts as a bridge between the application and the database.
    - It includes DbSet<TEntity> properties for each entity type that the context manages.

2. DbSet<TEntity>

- **DbSet<TEntity>**:
    - Represents a collection of entities of a specific type that can be queried from the database.
    - Provides methods to perform CRUD (Create, Read, Update, Delete) operations.

3. OnModelCreating Method

- **OnModelCreating**:
    - This method is used to configure the model and relationships using Fluent API. It is where you define custom configurations and constraints.

Code Walkthrough

ApplicationDbContext Class

```
using eCommerce.DataAccessLayer.Entities;
using Microsoft.EntityFrameworkCore;

namespace eCommerce.DataAccessLayer.Context;

public class ApplicationDbContext : DbContext
{
  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options)
  {
  }

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

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    base.OnModelCreating(modelBuilder);
  }
}
```

- **Constructor**:

  ```
  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options)
  {
  }
  ```

  - **Purpose**: Initializes a new instance of ApplicationDbContext with the specified options. DbContextOptions<ApplicationDbContext> provides configuration settings for the context, such as the database provider and connection string.

- **DbSet<Product>**:

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

  - **Purpose**: Represents the collection of Product entities in the context. This property allows querying and saving Product entities to the Products table in the database.

- **OnModelCreating Method**:

  ```
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    base.OnModelCreating(modelBuilder);
  }
  ```

  - **Purpose**: Configures the model and relationships using Fluent API. By calling base.OnModelCreating(modelBuilder), it ensures that any configurations from the base class are also applied. This method is currently empty but can be used to add custom configurations, such as entity relationships or constraints, in the future.

Summary of Key Concepts of DbContext

1. **DbContext**:
   - ApplicationDbContext is responsible for managing the connection to the database and providing access to Product entities.
2. **DbSet<TEntity>**:
   - DbSet<Product> allows querying and manipulating Product entities in the Products table.
3. **OnModelCreating**:
   - Used to configure the model and relationships, allowing for custom configurations if needed.

The ApplicationDbContext class serves as the main interface for interacting with the database in the Products microservice, providing the necessary methods and properties for managing Product entities.

Repository Contract in DataAccessLayer

The IProductsRepository interface defines the contract for accessing and manipulating Product data in the Products table. It specifies various methods for CRUD operations and querying the database.

Concepts

1. Repository Pattern

- **Repository Pattern**:
  - A design pattern that abstracts the data access logic and provides a simple interface for CRUD operations. It helps in managing data retrieval and storage, promoting separation of concerns and testability.

2. Asynchronous Programming

- **Asynchronous Programming**:
  - Techniques that allow operations to run in the background without blocking the main thread. Methods returning Task or Task<T> are typically used to perform non-blocking I/O operations.

3. Expressions

- **Expression<Func<T, bool>>**:
  - Represents a lambda expression that can be compiled into a query. It allows dynamic filtering of data based on conditions.

Code Walkthrough

IProductsRepository Interface

```
using eCommerce.DataAccessLayer.Entities;

using System.Linq.Expressions;


namespace eCommerce.DataAccessLayer.RepositoryContracts;


/// <summary>

/// Represents a repository for managing 'products' table

/// </summary>

public interface IProductsRepository
```

```
{
  /// <summary>
  /// Retrieves all products asynchronously
  /// </summary>
  /// <returns>Returns all products from the table</returns>
  Task<IEnumerable<Product>> GetProducts();

  /// <summary>
  /// Retrieves all products based on the specified condition asynchronously.
  /// </summary>
  /// <param name="conditionExpression">The condition to filter products</param>
  /// <returns>Returning a collection of matching products</returns>
  Task<IEnumerable<Product?>> GetProductsByCondition(Expression<Func<Product, bool>>
conditionExpression);

  /// <summary>
  /// Retrieves a single product based on the specified condition asynchronously
  /// </summary>
  /// <param name="conditionExpression">The condition to filter the product</param>
  /// <returns>Returns a single product or null if not found</returns>
  Task<Product?> GetProductByCondition(Expression<Func<Product, bool>> conditionExpression);

  /// <summary>
  /// Adds a new product into the products table asynchronously
  /// </summary>
  /// <param name="product">The product to be added</param>
  /// <returns>Returns the added product object or null if unsuccessful</returns>
  Task<Product?> AddProduct(Product product);

  /// <summary>
  /// Updates an existing product asynchronously.
```

```
 /// </summary>
 /// <param name="product">The product to be updated</param>
 /// <returns>Returns the updated product; or null if not found</returns>
 Task<Product?> UpdateProduct(Product product);


 /// <summary>
 /// Deletes the product asynchronously
 /// </summary>
 /// <param name="productID">The product ID to be deleted</param>
 /// <returns>Returns true if the deletion is successful, false otherwise.</returns>
 Task<bool> DeleteProduct(Guid productID);
}
```

- **GetProducts Method**:

  Task<IEnumerable<Product>> GetProducts();

  - **Purpose**: Asynchronously retrieves all Product entities from the database. Returns a collection of products.

- **GetProductsByCondition Method**:

  Task<IEnumerable<Product?>> GetProductsByCondition(Expression<Func<Product, bool>> conditionExpression);

  - **Purpose**: Asynchronously retrieves products based on a condition specified by the conditionExpression. Returns a collection of products that match the condition.

- **GetProductByCondition Method**:

  Task<Product?> GetProductByCondition(Expression<Func<Product, bool>> conditionExpression);

  - **Purpose**: Asynchronously retrieves a single product based on the conditionExpression. Returns a single product or null if no product matches the condition.

- **AddProduct Method**:

  Task<Product?> AddProduct(Product product);

  - **Purpose**: Asynchronously adds a new Product entity to the database. Returns the added product object or null if the operation was unsuccessful.

- **UpdateProduct Method**:

  Task<Product?> UpdateProduct(Product product);

  - **Purpose**: Asynchronously updates an existing Product entity in the database. Returns the updated product or null if the product was not found.

- **DeleteProduct Method**:

  Task<bool> DeleteProduct(Guid productID);

  - **Purpose**: Asynchronously deletes a Product entity based on the provided productID. Returns true if the deletion was successful and false otherwise.

Summary of Key Concepts of Repository Contract

1. **Repository Pattern**:
   - Abstracts data access logic, providing a clear interface for CRUD operations.
2. **Asynchronous Programming**:
   - Uses Task to perform non-blocking I/O operations, improving performance and responsiveness.
3. **Expressions**:
   - Allows dynamic querying of data based on runtime conditions, facilitating flexible and powerful data access.

The IProductsRepository interface outlines the necessary methods for managing Product entities, promoting a clean separation between data access and business logic.

Repository Implementation in DataAccessLayer

The ProductsRepository class implements the IProductsRepository interface, providing concrete methods for accessing and manipulating Product data using Entity Framework Core. It interacts with the database through the ApplicationDbContext context.

Code Walkthrough

ProductsRepository Class

```
using eCommerce.DataAccessLayer.Entities;
using eCommerce.DataAccessLayer.Context;
using eCommerce.DataAccessLayer.RepositoryContracts;
using Microsoft.EntityFrameworkCore;
using System.Linq.Expressions;

namespace eCommerce.DataAccessLayer.Repositories;

public class ProductsRepository : IProductsRepository
{
  private readonly ApplicationDbContext _dbContext;

  public ProductsRepository(ApplicationDbContext dbContext)
  {
    _dbContext = dbContext;
  }

  public async Task<Product?> AddProduct(Product product)
  {
    _dbContext.Products.Add(product);
    await _dbContext.SaveChangesAsync();
    return product;
  }

  public async Task<bool> DeleteProduct(Guid productID)
```

```
 {
   Product? existingProduct = await _dbContext.Products.FirstOrDefaultAsync(temp =>
temp.ProductID == productID);
   if (existingProduct == null)
   {
     return false;
   }

   _dbContext.Products.Remove(existingProduct);
   int affectedRowsCount = await _dbContext.SaveChangesAsync();
   return affectedRowsCount > 0;
 }

 public async Task<Product?> GetProductByCondition(Expression<Func<Product, bool>>
conditionExpression)
 {
   return await _dbContext.Products.FirstOrDefaultAsync(conditionExpression);
 }

 public async Task<IEnumerable<Product>> GetProducts()
 {
   return await _dbContext.Products.ToListAsync();
 }

 public async Task<IEnumerable<Product?>> GetProductsByCondition(Expression<Func<Product,
bool>> conditionExpression)
 {
   return await _dbContext.Products.Where(conditionExpression).ToListAsync();
 }

 public async Task<Product?> UpdateProduct(Product product)
 {
   Product? existingProduct = await _dbContext.Products.FirstOrDefaultAsync(temp =>
temp.ProductID == product.ProductID);
   if (existingProduct == null)
   {
     return null;
   }

   existingProduct.ProductName = product.ProductName;
   existingProduct.UnitPrice = product.UnitPrice;
   existingProduct.QuantityInStock = product.QuantityInStock;
   existingProduct.Category = product.Category;

   await _dbContext.SaveChangesAsync();

   return existingProduct;
 }
}
```

Key Methods

1. **AddProduct**

   ```
   public async Task<Product?> AddProduct(Product product)
   {
     _dbContext.Products.Add(product);
     await _dbContext.SaveChangesAsync();
     return product;
   }
   ```

   - **Purpose**: Adds a new product to the Products table and saves the changes to the database. Returns the added product.

2. **DeleteProduct**

   ```
   public async Task<bool> DeleteProduct(Guid productID)
   {
   ```

```
  Product? existingProduct = await _dbContext.Products.FirstOrDefaultAsync(temp =>
temp.ProductID == productID);
  if (existingProduct == null)
  {
    return false;
  }

  _dbContext.Products.Remove(existingProduct);
  int affectedRowsCount = await _dbContext.SaveChangesAsync();
  return affectedRowsCount > 0;
}
```

   - **Purpose**: Deletes a product with the specified productID. Returns true if the deletion was successful, otherwise false.

3. **GetProductByCondition**

```
public async Task<Product?> GetProductByCondition(Expression<Func<Product, bool>>
conditionExpression)
{
  return await _dbContext.Products.FirstOrDefaultAsync(conditionExpression);
}
```

   - **Purpose**: Retrieves a single product that matches the specified condition. Returns the product or null if not found.

4. **GetProducts**

```
public async Task<IEnumerable<Product>> GetProducts()
{
  return await _dbContext.Products.ToListAsync();
}
```

   - **Purpose**: Retrieves all products from the Products table. Returns a collection of all products.

5. **GetProductsByCondition**

```
public async Task<IEnumerable<Product?>>
GetProductsByCondition(Expression<Func<Product, bool>> conditionExpression)
{
  return await _dbContext.Products.Where(conditionExpression).ToListAsync();
}
```

   - **Purpose**: Retrieves products that match the specified condition. Returns a collection of matching products.

6. **UpdateProduct**

```
public async Task<Product?> UpdateProduct(Product product)
{
  Product? existingProduct = await _dbContext.Products.FirstOrDefaultAsync(temp =>
temp.ProductID == product.ProductID);
  if (existingProduct == null)
  {
    return null;
  }

  existingProduct.ProductName = product.ProductName;
  existingProduct.UnitPrice = product.UnitPrice;
  existingProduct.QuantityInStock = product.QuantityInStock;
  existingProduct.Category = product.Category;

  await _dbContext.SaveChangesAsync();

  return existingProduct;
}
```

   - **Purpose**: Updates an existing product. Returns the updated product or null if the product was not found.

Summary of Repository Implementation

- **CRUD Operations**: ProductsRepository implements methods for creating, reading, updating, and deleting Product entities using Entity Framework Core.
- **Asynchronous Programming**: Methods use Task to perform non-blocking database operations, ensuring responsiveness and scalability.
- **Entity Framework Core**: Utilizes DbContext and LINQ to query and manipulate data, leveraging the capabilities of EF Core for ORM (Object-Relational Mapping).

Dependency Injection in DataAccessLayer

The DependencyInjection class in the DataAccessLayer is responsible for registering data access services with the IoC (Inversion of Control) container. This includes configuring the database context and repository services.

Here's a detailed look at the code and its purpose:

Code Walkthrough

```
using eCommerce.DataAccessLayer.Context;
using eCommerce.DataAccessLayer.Repositories;
using eCommerce.DataAccessLayer.RepositoryContracts;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace eCommerce.ProductsService.DataAccessLayer;

public static class DependencyInjection
{
  public static IServiceCollection AddDataAccessLayer(this IServiceCollection services, IConfiguration configuration)
  {
    // TO DO: Add Data Access Layer services into the IoC container
    // Configure DbContext to use MySQL with the connection string from configuration
    services.AddDbContext<ApplicationDbContext>(options => {
      options.UseMySQL(configuration.GetConnectionString("DefaultConnection")!);
    });

    // Register repository with a scoped lifetime
    services.AddScoped<IProductsRepository, ProductsRepository>();

    return services;
  }
}
```

Key Components

1. **DbContext Configuration**

   ```
   services.AddDbContext<ApplicationDbContext>(options => {
     options.UseMySQL(configuration.GetConnectionString("DefaultConnection")!);
   });
   ```

   - **Purpose**: Configures ApplicationDbContext to use MySQL as the database provider. It retrieves the connection string named "DefaultConnection" from the application's configuration settings.
   - **Method**: AddDbContext<TContext> registers the ApplicationDbContext with the dependency injection container and configures it to use MySQL. Note that UseMySQL is used here, but if you are using PostgreSQL, you would typically use UseNpgsql.

2. **Repository Registration**

   ```
   services.AddScoped<IProductsRepository, ProductsRepository>();
   ```

   - **Purpose**: Registers the ProductsRepository class as the implementation of the IProductsRepository interface.
   - **Lifetime**: Scoped lifetime ensures that a single instance of the repository is used within a single request scope, which is suitable for managing database transactions.

Usage

To use the AddDataAccessLayer method, you would typically call it from the Startup.cs or Program.cs file of your application:

```
public class Startup

{

  public void ConfigureServices(IServiceCollection services)

  {

    // Other service registrations


    services.AddDataAccessLayer(Configuration);

  }

}
```

Summary of Dependency Injection DataAccessLayer

- **DbContext Configuration**: Registers ApplicationDbContext with MySQL configuration, allowing it to interact with the database.
- **Repository Registration**: Adds ProductsRepository as the implementation of IProductsRepository, ensuring it is available for dependency injection.
- **Scoped Lifetime**: Ensures that the repository and DbContext are created and disposed of within the same request scope, promoting efficient resource management.

DTOs in the Business Logic Layer

The Data Transfer Objects (DTOs) in the Business Logic Layer are used to encapsulate data for various operations related to Product. These DTOs facilitate the transfer of data between different layers of the application, such as between the API layer and the business logic layer.

Here's a breakdown of each DTO:

1. CategoryOptions Enum

```
namespace eCommerce.BusinessLogicLayer.DTO;


public enum CategoryOptions

{

  Electronics, HomeAppliances, Furniture, Accessories

}
```

- **Purpose**: Defines the categories that products can belong to. This enum provides a set of predefined values that help in categorizing products.

2. ProductAddRequest Record

```
namespace eCommerce.BusinessLogicLayer.DTO;


public record ProductAddRequest(string ProductName, CategoryOptions Category, double?
UnitPrice, int? QuantityInStock)

{

  public ProductAddRequest() : this(default, default, default, default)

  {

  }

}
```

- **Purpose**: Represents the data required to add a new product.
- **Properties**:
    - ProductName: The name of the product.
    - Category: The category of the product, using the CategoryOptions enum.
    - UnitPrice: The price of the product.
    - QuantityInStock: The quantity available in stock.

- **Constructor**: Includes a parameterless constructor that sets all properties to their default values, useful for scenarios where default initialization is required.

3. ProductResponse Record

```
namespace eCommerce.BusinessLogicLayer.DTO;


public record ProductResponse(Guid ProductID, string ProductName, CategoryOptions Category, double? UnitPrice, int? QuantityInStock)
{
  public ProductResponse() : this(default, default, default, default, default)
  {
  }
}
```

- **Purpose**: Represents the data returned when querying for a product. This DTO is used to return the details of a product.
- **Properties**:
  - ProductID: The unique identifier of the product.
  - ProductName: The name of the product.
  - Category: The category of the product.
  - UnitPrice: The price of the product.
  - QuantityInStock: The quantity available in stock.
- **Constructor**: Includes a parameterless constructor for default initialization.

4. ProductUpdateRequest Record

```
namespace eCommerce.BusinessLogicLayer.DTO;


public record ProductUpdateRequest(Guid ProductID, string ProductName, CategoryOptions Category, double? UnitPrice, int? QuantityInStock)
{
  public ProductUpdateRequest() : this(default, default, default, default, default)
  {
  }
}
```

- **Purpose**: Represents the data required to update an existing product.
- **Properties**:
  - ProductID: The unique identifier of the product being updated.
  - ProductName: The name of the product.
  - Category: The category of the product.
  - UnitPrice: The price of the product.
  - QuantityInStock: The quantity available in stock.
- **Constructor**: Includes a parameterless constructor for default initialization.

Summary of DTO

- **Enums and Records**: Enums define fixed sets of values (e.g., CategoryOptions), while records are used for encapsulating data with immutable properties (e.g., ProductAddRequest, ProductResponse, ProductUpdateRequest).
- **Default Constructors**: Each DTO includes a parameterless constructor for cases where default values are needed.

These DTOs facilitate the interaction between different layers by providing structured and typed data for product-related operations.

Service Contracts in the Business Logic Layer

The service contracts define the operations that the business logic layer will expose for managing products. These operations interact with the data access layer and convert data into the format required by the API layer or other consumers.

Here's a breakdown of each service contract:

1. IProductsService Interface

```
using eCommerce.DataAccessLayer.Entities;
using eCommerce.BusinessLogicLayer.DTO;
using System.Linq.Expressions;

namespace eCommerce.BusinessLogicLayer.ServiceContracts;

public interface IProductsService
{
  /// <summary>
  /// Retrieves the list of products from the products repository.
  /// </summary>
  /// <returns>Returns list of ProductResponse objects.</returns>
  Task<List<ProductResponse?>> GetProducts();

  /// <summary>
  /// Retrieves list of products matching the given condition.
  /// </summary>
  /// <param name="conditionExpression">Expression that represents the condition to
check.</param>
  /// <returns>Returns matching products.</returns>
  Task<List<ProductResponse?>> GetProductsByCondition(Expression<Func<Product, bool>>
conditionExpression);

  /// <summary>
  /// Returns a single product that matches the given condition.
  /// </summary>
  /// <param name="conditionExpression">Expression that represents the condition to
check.</param>
  /// <returns>Returns matching product or null.</returns>
  Task<ProductResponse?> GetProductByCondition(Expression<Func<Product, bool>>
conditionExpression);


  /// <summary>
  /// Adds (inserts) a product into the table using the products repository.
  /// </summary>
  /// <param name="productAddRequest">Product to insert.</param>
  /// <returns>Product after inserting or null if unsuccessful.</returns>
  Task<ProductResponse?> AddProduct(ProductAddRequest productAddRequest);


  /// <summary>
  /// Updates the existing product based on the ProductID.
  /// </summary>
  /// <param name="productUpdateRequest">Product data to update.</param>
  /// <returns>Returns product object after successful updation; otherwise null.</returns>
  Task<ProductResponse?> UpdateProduct(ProductUpdateRequest productUpdateRequest);


  /// <summary>
  /// Deletes an existing product based on the given product ID.
  /// </summary>
  /// <param name="productID">ProductID to search and delete.</param>
  /// <returns>Returns true if the deletion is successful; otherwise false.</returns>
  Task<bool> DeleteProduct(Guid productID);
}
```

Summary of Service Contract Methods:

1. **GetProducts**:
   - **Purpose**: Retrieves all products from the repository.
   - **Returns**: A list of ProductResponse objects.
2. **GetProductsByCondition**:
   - **Purpose**: Retrieves products that match a specified condition.
   - **Parameters**: conditionExpression - An expression representing the filter condition.
   - **Returns**: A list of ProductResponse objects matching the condition.
3. **GetProductByCondition**:
   - **Purpose**: Retrieves a single product that matches a specified condition.
   - **Parameters**: conditionExpression - An expression representing the filter condition.
   - **Returns**: A single ProductResponse object or null if no product matches.
4. **AddProduct**:
   - **Purpose**: Adds a new product to the repository.
   - **Parameters**: productAddRequest - Data required to create a new product.
   - **Returns**: The added ProductResponse object or null if the addition was unsuccessful.
5. **UpdateProduct**:
   - **Purpose**: Updates an existing product in the repository.
   - **Parameters**: productUpdateRequest - Data required to update an existing product.
   - **Returns**: The updated ProductResponse object or null if the update was unsuccessful.
6. **DeleteProduct**:
   - **Purpose**: Deletes a product from the repository based on its ID.
   - **Parameters**: productID - The ID of the product to be deleted.
   - **Returns**: True if the deletion was successful, otherwise false.

Usage

- **Data Access**: These service methods interact with the data access layer (IProductsRepository) to perform CRUD operations.
- **Business Logic**: The business logic layer applies additional logic or transformations on the data as needed.

- **API Layer**: The API layer consumes these services to expose functionality through HTTP endpoints.

By defining these contracts, the business logic layer provides a clear API for managing products, ensuring that the data access and presentation layers remain decoupled.

Mappers in the Business Logic Layer

The mappers in the Business Logic Layer use AutoMapper to map between Data Transfer Objects (DTOs) and entities. This approach ensures that the application's different layers remain decoupled and that data is transformed correctly as it flows through the system.

1. ProductAddRequest to Product Mapping Profile

This profile handles the mapping from ProductAddRequest (DTO) to Product (entity) for when new products are added.

```
using AutoMapper;

using eCommerce.DataAccessLayer.Entities;

using eCommerce.BusinessLogicLayer.DTO;


namespace eCommerce.BusinessLogicLayer.Mappers;


public class ProductAddRequestToProductMappingProfile : Profile

{

  public ProductAddRequestToProductMappingProfile()

  {

    CreateMap<ProductAddRequest, Product>()

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

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

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

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

      .ForMember(dest => dest.ProductID, opt => opt.Ignore()); // ProductID is ignored during addition

  }

}
```

- **Purpose**: Converts the ProductAddRequest DTO to a Product entity when a new product is being created.
- **Key Points**:
    - Ignores ProductID as it will be generated during the insert operation.

2. Product to ProductResponse Mapping Profile

This profile maps the Product entity to the ProductResponse DTO for returning product details to the client.

```
using AutoMapper;

using eCommerce.DataAccessLayer.Entities;

using eCommerce.BusinessLogicLayer.DTO;


namespace eCommerce.BusinessLogicLayer.Mappers;


public class ProductToProductResponseMappingProfile : Profile

{

  public ProductToProductResponseMappingProfile()

  {

    CreateMap<Product, ProductResponse>()

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

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

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

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

      .ForMember(dest => dest.ProductID, opt => opt.MapFrom(src => src.ProductID)); // Maps the ProductID

  }

}
```

- **Purpose**: Converts the Product entity into a ProductResponse DTO for output to the client.
- **Key Points**:
    - All fields, including ProductID, are directly mapped.

3. ProductUpdateRequest to Product Mapping Profile

This profile maps from the ProductUpdateRequest DTO to the Product entity for updating existing products.

```
using AutoMapper;

using eCommerce.DataAccessLayer.Entities;

using eCommerce.BusinessLogicLayer.DTO;


namespace eCommerce.BusinessLogicLayer.Mappers;


public class ProductUpdateRequestToProductMappingProfile : Profile

{

  public ProductUpdateRequestToProductMappingProfile()

  {

    CreateMap<ProductUpdateRequest, Product>()

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

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

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

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

      .ForMember(dest => dest.ProductID, opt => opt.MapFrom(src => src.ProductID)); // Maps ProductID for the update

  }

}
```

- **Purpose**: Converts the ProductUpdateRequest DTO to a Product entity for updates.
- **Key Points**:
    - All fields, including ProductID, are directly mapped, which is important for identifying the product to update.

Summary of Usage of Mappers in BusinessLogicLayer

1. **ProductAddRequestToProductMappingProfile**:
   - Maps from ProductAddRequest DTO to the Product entity during product addition.
2. **ProductToProductResponseMappingProfile**:
   - Maps from Product entity to ProductResponse DTO when returning product details to clients.
3. **ProductUpdateRequestToProductMappingProfile**:
   - Maps from ProductUpdateRequest DTO to the Product entity when updating product details.

These mappings allow the service layer to handle complex transformations in a clean and efficient way, abstracting the data transformations between different layers of the application.

Validators in the Business Logic Layer

The validators ensure that the data passed to the system through ProductAddRequest and ProductUpdateRequest is valid and adheres to the necessary business rules before further processing occurs. FluentValidation is used for this purpose, providing a fluent interface to define validation rules.

1. ProductAddRequestValidator

This validator is used for validating ProductAddRequest DTO, ensuring the product creation request meets specific criteria.

```
using eCommerce.BusinessLogicLayer.DTO;

using FluentValidation;


namespace eCommerce.BusinessLogicLayer.Validators;


public class ProductAddRequestValidator : AbstractValidator<ProductAddRequest>

{

  public ProductAddRequestValidator()

  {
   //ProductName
   RuleFor(temp => temp.ProductName)
    .NotEmpty().WithMessage("Product Name can't be blank");

   //Category
   RuleFor(temp => temp.Category)
    .IsInEnum().WithMessage("Category can't be blank");

   //UnitPrice
   RuleFor(temp => temp.UnitPrice)
    .InclusiveBetween(0, double.MaxValue).WithMessage($"Unit Price should be between 0 and {double.MaxValue}");

   //QuantityInStock
   RuleFor(temp => temp.QuantityInStock)
    .InclusiveBetween(0, int.MaxValue).WithMessage($"Quantity in Stock should be between 0 and {int.MaxValue}");
  }
}
```

- **Key Validations**:
    - **ProductName**: Cannot be empty.
    - **Category**: Must be one of the predefined enum values.
    - **UnitPrice**: Must be a positive value between 0 and double.MaxValue.
    - **QuantityInStock**: Must be a positive integer between 0 and int.MaxValue.

2. ProductUpdateRequestValidator

This validator is used for validating the ProductUpdateRequest DTO, ensuring the product update request is correct before updating an existing product.

```
using eCommerce.BusinessLogicLayer.DTO;

using FluentValidation;


namespace eCommerce.BusinessLogicLayer.Validators;


public class ProductUpdateRequestValidator : AbstractValidator<ProductUpdateRequest>

{

  public ProductUpdateRequestValidator()

  {

    //ProductID

    RuleFor(temp => temp.ProductID)

      .NotEmpty().WithMessage("Product ID can't be blank");


    //ProductName

    RuleFor(temp => temp.ProductName)

      .NotEmpty().WithMessage("Product Name can't be blank");


    //Category

    RuleFor(temp => temp.Category)

      .IsInEnum().WithMessage("Category can't be blank");


    //UnitPrice

    RuleFor(temp => temp.UnitPrice)

      .InclusiveBetween(0, double.MaxValue).WithMessage($"Unit Price should be between 0 and {double.MaxValue}");


    //QuantityInStock

    RuleFor(temp => temp.QuantityInStock)
    .InclusiveBetween(0, int.MaxValue).WithMessage($"Quantity in Stock should be between 0 and {int.MaxValue}");
  }
}
```

- **Key Validations**:
    - **ProductID**: Must not be empty to identify the product to update.
    - **ProductName**: Cannot be empty.
    - **Category**: Must be a valid enum value.
    - **UnitPrice**: Must be between 0 and double.MaxValue.
    - **QuantityInStock**: Must be a positive integer between 0 and int.MaxValue.

Summary of Validators in the Business Logic Layer

1. **ProductAddRequestValidator**: Validates that the required fields for adding a product, such as product name, category, unit price, and quantity in stock, are provided and valid.
2. **ProductUpdateRequestValidator**: Ensures that when updating a product, the product ID, along with other necessary fields like name, category, unit price, and quantity, are correct and valid.

These validators enforce the necessary business rules before data is passed further into the service layer or repository layer.

Services in BusinessLogicLayer

The ProductsService class provides the business logic for interacting with product-related operations. It sits between the data access layer (repositories) and external requests, applying validation, mapping, and additional business rules.

Key Components:

- **Validators**: Uses FluentValidation for validating incoming requests (e.g., ProductAddRequest and ProductUpdateRequest).
- **Mapper**: AutoMapper is used to map between the request DTOs (ProductAddRequest, ProductUpdateRequest) and entity models (Product) and also between entities and response DTOs (ProductResponse).
- **Repository**: Interacts with the repository layer to handle data access operations.

Constructor

The constructor receives dependencies via Dependency Injection:

```
public ProductsService(
   IValidator<ProductAddRequest> productAddRequestValidator,
   IValidator<ProductUpdateRequest> productUpdateRequestValidator,
   IMapper mapper,
   IProductsRepository productsRepository)
{
   _productAddRequestValidator = productAddRequestValidator;
   _productUpdateRequestValidator = productUpdateRequestValidator;
   _mapper = mapper;
   _productsRepository = productsRepository;
}
```

Methods Overview

1. AddProduct

Adds a new product to the repository after validation.

```
public async Task<ProductResponse?> AddProduct(ProductAddRequest productAddRequest)
{
   if (productAddRequest == null) throw new ArgumentNullException(nameof(productAddRequest));


   ValidationResult validationResult = await _productAddRequestValidator.ValidateAsync(productAddRequest);
   if (!validationResult.IsValid)
   {
      string errors = string.Join(", ", validationResult.Errors.Select(temp => temp.ErrorMessage));
      throw new ArgumentException(errors);
   }


   Product productInput = _mapper.Map<Product>(productAddRequest);
   Product? addedProduct = await _productsRepository.AddProduct(productInput);


   if (addedProduct == null) return null;


   ProductResponse addedProductResponse = _mapper.Map<ProductResponse>(addedProduct);
   return addedProductResponse;
}
```

- **Validation**: Ensures the incoming ProductAddRequest is valid.
- **Mapping**: Maps ProductAddRequest to Product entity using AutoMapper.
- **Repository Interaction**: Calls the AddProduct method in the repository.
- **Response**: Maps the added Product entity to a ProductResponse.

2. DeleteProduct

Deletes a product by its ID.

```
public async Task<bool> DeleteProduct(Guid productID)
{
   Product? existingProduct = await _productsRepository.GetProductByCondition(temp =>
temp.ProductID == productID);
   if (existingProduct == null) return false;

   bool isDeleted = await _productsRepository.DeleteProduct(productID);
   return isDeleted;
}
```

- **Product Existence**: Checks if the product exists before attempting deletion.
- **Repository Interaction**: Deletes the product by ID if found.

3. GetProductByCondition

Retrieves a product based on a condition (e.g., by ID or other criteria).

```
public async Task<ProductResponse?> GetProductByCondition(Expression<Func<Product,
bool>> conditionExpression)
{
   Product? product = await
_productsRepository.GetProductByCondition(conditionExpression);
   if (product == null) return null;

   ProductResponse productResponse = _mapper.Map<ProductResponse>(product);
   return productResponse;
}
```

- **Repository Interaction**: Fetches a product based on a condition.
- **Mapping**: Maps the fetched Product to a ProductResponse.

4. GetProducts

Retrieves a list of all products.

```
public async Task<List<ProductResponse?>> GetProducts()
```

```
{
   IEnumerable<Product?> products = await _productsRepository.GetProducts();
   IEnumerable<ProductResponse?> productResponses =
_mapper.Map<IEnumerable<ProductResponse>>(products);
   return productResponses.ToList();
}
```

- **Repository Interaction**: Retrieves all products from the repository.
- **Mapping**: Maps the collection of Product entities to a list of ProductResponse objects.

5. GetProductsByCondition

Retrieves a list of products based on a condition (e.g., category or stock availability).

```
public async Task<List<ProductResponse?>>
GetProductsByCondition(Expression<Func<Product, bool>> conditionExpression)
{
   IEnumerable<Product?> products = await
_productsRepository.GetProductsByCondition(conditionExpression);
   IEnumerable<ProductResponse?> productResponses =
_mapper.Map<IEnumerable<ProductResponse>>(products);
   return productResponses.ToList();
}
```

- **Repository Interaction**: Fetches products matching the specified condition.
- **Mapping**: Maps the collection of Product entities to ProductResponse.

6. UpdateProduct

Updates an existing product based on the provided request.

```
public async Task<ProductResponse?> UpdateProduct(ProductUpdateRequest
productUpdateRequest)
{
   Product? existingProduct = await _productsRepository.GetProductByCondition(temp =>
temp.ProductID == productUpdateRequest.ProductID);
   if (existingProduct == null) throw new ArgumentException("Invalid Product ID");
   ValidationResult validationResult = await
_productUpdateRequestValidator.ValidateAsync(productUpdateRequest);

   if (!validationResult.IsValid)

   {

      string errors = string.Join(", ", validationResult.Errors.Select(temp =>
temp.ErrorMessage));

      throw new ArgumentException(errors);

   }


   Product product = _mapper.Map<Product>(productUpdateRequest);

   Product? updatedProduct = await _productsRepository.UpdateProduct(product);

   ProductResponse? updatedProductResponse =
_mapper.Map<ProductResponse>(updatedProduct);


   return updatedProductResponse;

}
```

- **Validation**: Validates the update request.
- **Repository Interaction**: Updates the product if valid.
- **Mapping**: Maps both request and response between DTO and entity.

Summary of Services in BusinessLogicLayer

The ProductsService ensures that data is properly validated and mapped before interacting with the repository. It provides all essential product-related business operations such as adding, updating, retrieving, and deleting products, leveraging validation through FluentValidation and mapping through AutoMapper.

DependencyInjection in BusinessLogicLayer

This DependencyInjection class configures the services required for the Business Logic Layer (BLL) and registers them in the application's IoC container.

Key Components

- **AutoMapper**: Registers the mapping profiles used for converting between DTOs and entities.
- **FluentValidation**: Adds validators for handling validation of incoming requests.
- **ProductsService**: Registers the ProductsService to be used whenever the IProductsService contract is requested.

Implementation

```
using eCommerce.BusinessLogicLayer.Mappers;

using eCommerce.BusinessLogicLayer.ServiceContracts;

using Microsoft.Extensions.DependencyInjection;

using eCommerce.BusinessLogicLayer.Validators;

using FluentValidation;


namespace eCommerce.ProductsService.BusinessLogicLayer

{

  public static class DependencyInjection

  {

    public static IServiceCollection AddBusinessLogicLayer(this IServiceCollection services)

    {

      // Register AutoMapper with all mapping profiles from the assembly

services.AddAutoMapper(typeof(ProductAddRequestToProductMappingProfile).Assembly);


      // Register FluentValidation validators from the same assembly as
ProductAddRequestValidator

      services.AddValidatorsFromAssemblyContaining<ProductAddRequestValidator>();


      // Register ProductsService with the IProductsService interface
    services.AddScoped<IProductsService,
eCommerce.BusinessLogicLayer.Services.ProductsService>();


    return services;
   }
 }
}
```

Service Registrations

1. **AutoMapper**:
   - The AddAutoMapper() method registers the mapping profiles in the specified assembly, enabling the use of AutoMapper for object transformation.
   - All profiles in the assembly containing ProductAddRequestToProductMappingProfile are scanned and added to the IoC container.
2. **FluentValidation**:
   - The AddValidatorsFromAssemblyContaining<ProductAddRequestValidator>() method registers all validators in the assembly where ProductAddRequestValidator exists.
   - This allows any request DTO to be validated using the registered validators.
3. **ProductsService**:
   - Registers the ProductsService class as the implementation of the IProductsService interface.
   - This ensures that when IProductsService is requested, an instance of ProductsService is injected.

Summary of DependencyInjection in BusinessLogicLayer

This DependencyInjection class configures the necessary components of the Business Logic Layer, ensuring that:

- Mapping profiles and validators are registered and available.
- The business service (ProductsService) is accessible via dependency injection through its interface (IProductsService).

This setup allows for a clean separation of concerns and scalable service registration in the ASP.NET Core application.

appsettings.json in API Layer

The appsettings.json file contains configuration settings for the application, including logging, allowed hosts, and connection strings.

Configuration Structure

```
{
 "Logging": {
  "LogLevel": {
   "Default": "Information",
   "Microsoft.AspNetCore": "Warning"
  }
 },
 "AllowedHosts": "*",
 "ConnectionStrings": {
  "DefaultConnection": "Server=localhost; Port=3306; Database=ecommerceproductsdatabase; User ID=root; Password=admin"
 }
}
```

Explanation of Key Sections

1. **Logging**:
   - Defines the log levels for the application.
   - Default is set to Information to log general information messages.
   - Microsoft.AspNetCore is set to Warning, which logs warnings and above for the ASP.NET Core infrastructure.
2. **AllowedHosts**:
   - Set to "*", allowing the application to accept requests from any host.
   - In production, you can specify allowed host names for security.

3. **ConnectionStrings**:
    - Defines the DefaultConnection string for connecting to the MySQL database.
    - The connection string includes:
        - Server=localhost: The database is hosted locally.
        - Port=3306: The port MySQL uses.
        - Database=ecommerceproductsdatabase: The name of the database.
        - User ID=root: The root username for authentication.
        - Password=admin: The password for the database user.

Usage in Code

This file is referenced in the Program.cs or Startup.cs to configure logging and database connection for the API.

Example of retrieving the connection string in code:

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

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

{

   options.UseMySQL(connectionString);

});
```

Summary of appsettings.json in API Layer

This appsettings.json file handles key configuration settings for the API, including logging, host settings, and database connection strings, making it easier to manage and modify configuration without hardcoding values.

ExceptionHandlingMiddleware in the API Layer

This ExceptionHandlingMiddleware is a custom middleware for handling exceptions in the API layer of your application. It catches any unhandled exceptions during the request pipeline, logs the error details, and returns a JSON response with the error message.

Code Explanation

Middleware Class (ExceptionHandlingMiddleware)

```
namespace eCommerce.ProductsMicroService.API.Middleware;


public class ExceptionHandlingMiddleware
{
  private readonly RequestDelegate _next;
  private readonly ILogger<ExceptionHandlingMiddleware> _logger;


  public ExceptionHandlingMiddleware(RequestDelegate next,
ILogger<ExceptionHandlingMiddleware> logger)
  {
    _next = next;
    _logger = logger;
  }


  public async Task Invoke(HttpContext httpContext)
  {
    try
    {
      await _next(httpContext);
    }
    catch (Exception ex)
    {
      if (ex.InnerException != null)
      {
        _logger.LogError("{ExceptionType} {ExceptionMessage}",
ex.InnerException.GetType().ToString(), ex.InnerException.Message);
      }
      else
      {
        _logger.LogError("{ExceptionType} {ExceptionMessage}", ex.GetType().ToString(),
ex.Message);
      }

      // Return status code 500 for server error
      httpContext.Response.StatusCode = 500;

      // Write a JSON response with exception details
      await httpContext.Response.WriteAsJsonAsync(new {
        Message = ex.Message,
        Type = ex.GetType().ToString()
      });
    }
  }
}
```

Middleware Extension (ExceptionHandlingMiddlewareExtensions)

```
public static class ExceptionHandlingMiddlewareExtensions
{
  public static IApplicationBuilder UseExceptionHandlingMiddleware(this IApplicationBuilder
builder)
  {
    return builder.UseMiddleware<ExceptionHandlingMiddleware>();
  }
}
```

Key Features

1. **Logging**:
    - Catches all unhandled exceptions and logs their details using ILogger<ExceptionHandlingMiddleware>.
    - If the exception has an InnerException, it logs that as well.
2. **Error Response**:
    - The middleware sets the HTTP status code to 500 (Internal Server Error).
    - A JSON object is returned in the response, containing:
        - The exception message.
        - The exception type.
3. **Exception Handling**:
    - Catches and processes exceptions that occur in the request pipeline, ensuring no unhandled exceptions propagate further.

Usage in Program.cs

To activate the middleware, it must be added to the request pipeline in Program.cs or Startup.cs:

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


var app = builder.Build();


// Add the exception handling middleware

app.UseExceptionHandlingMiddleware();


app.Run();
```

Advantages of ExceptionHandlingMiddleware

- **Centralized Error Handling**: All exceptions are handled in one place, ensuring consistent behavior for unhandled exceptions.
- **Logging**: Detailed logging of exceptions helps in debugging and monitoring.
- **JSON Error Response**: Provides structured JSON responses that are easier for API clients to consume and process.

This middleware ensures graceful error handling, improved logging, and better error reporting in the API layer.

API Endpoints in the API Layer

The ProductAPIEndpoints class defines REST API endpoints for managing products in the eCommerce platform. These endpoints interact with the business logic layer to handle common product-related operations such as retrieving, adding, updating, and deleting products.

Endpoints Explanation

1. GET /api/products

Fetches all products.

```
app.MapGet("/api/products", async (IProductsService productsService) =>
{
   List<ProductResponse?> products = await productsService.GetProducts();
   return Results.Ok(products);
});
```

- **Description**: Retrieves all available products.
- **Returns**: A list of ProductResponse objects in the response.

2. GET /api/products/search/product-id/{ProductID}

Fetches a product by its unique ProductID.

```
app.MapGet("/api/products/search/product-id/{ProductID:guid}", async (IProductsService
productsService, Guid ProductID) =>
{
   ProductResponse? product = await productsService.GetProductByCondition(temp =>
temp.ProductID == ProductID);
   return Results.Ok(product);
});
```

- **Description**: Retrieves a product that matches the given ProductID.
- **Returns**: A single ProductResponse object or null if no matching product is found.

3. GET /api/products/search/{SearchString}

Searches for products by name or category.

```
app.MapGet("/api/products/search/{SearchString}", async (IProductsService productsService, string SearchString) =>
{
   List<ProductResponse?> productsByProductName = await productsService.GetProductsByCondition(temp => temp.ProductName != null && temp.ProductName.Contains(SearchString, StringComparison.OrdinalIgnoreCase));


   List<ProductResponse?> productsByCategory = await productsService.GetProductsByCondition(temp => temp.Category != null && temp.Category.Contains(SearchString, StringComparison.OrdinalIgnoreCase));


   var products = productsByProductName.Union(productsByCategory);


   return Results.Ok(products);
});
```

- **Description**: Searches for products by matching the SearchString with product names or categories.
- **Returns**: A list of matching products.

4. POST /api/products

Adds a new product.

```
app.MapPost("/api/products", async (IProductsService productsService, IValidator<ProductAddRequest> productAddRequestValidator, ProductAddRequest productAddRequest) =>
{
   // Validation using FluentValidation
   ValidationResult validationResult = await productAddRequestValidator.ValidateAsync(productAddRequest);


   if (!validationResult.IsValid)
   {
      Dictionary<string, string[]> errors = validationResult.Errors
```

```
            .GroupBy(temp => temp.PropertyName)
            .ToDictionary(grp => grp.Key, grp => grp.Select(err => err.ErrorMessage).ToArray());
        return Results.ValidationProblem(errors);
    }

    var addedProductResponse = await productsService.AddProduct(productAddRequest);
    if (addedProductResponse != null)
        return Results.Created($"/api/products/search/product-id/{addedProductResponse.ProductID}", addedProductResponse);
    else
        return Results.Problem("Error in adding product");
});
```

- **Description**: Adds a new product to the system.
- **Validates**: The product information using FluentValidation.
- **Returns**: The newly added product, or an error message if the operation fails.

5. PUT /api/products

Updates an existing product.

```
app.MapPut("/api/products", async (IProductsService productsService, IValidator<ProductUpdateRequest> productUpdateRequestValidator, ProductUpdateRequest productUpdateRequest) =>
{
    ValidationResult validationResult = await productUpdateRequestValidator.ValidateAsync(productUpdateRequest);

    if (!validationResult.IsValid)
    {
        Dictionary<string, string[]> errors = validationResult.Errors
            .GroupBy(temp => temp.PropertyName)
            .ToDictionary(grp => grp.Key, grp => grp.Select(err => err.ErrorMessage).ToArray());
        return Results.ValidationProblem(errors);
    }
```

```
  var updatedProductResponse = await productsService.UpdateProduct(productUpdateRequest);
  if (updatedProductResponse != null)
    return Results.Ok(updatedProductResponse);
  else
    return Results.Problem("Error in updating product");
});
```

- **Description**: Updates an existing product's details.
- **Validates**: The updated information using FluentValidation.
- **Returns**: The updated product details, or an error message if the operation fails.

6. DELETE /api/products/{ProductID}

Deletes a product by its ProductID.

```
app.MapDelete("/api/products/{ProductID:guid}", async (IProductsService productsService, Guid ProductID) =>
{
  bool isDeleted = await productsService.DeleteProduct(ProductID);
  if (isDeleted)
    return Results.Ok(true);
  else
    return Results.Problem("Error in deleting product");
});
```

- **Description**: Deletes the product that matches the provided ProductID.
- **Returns**: A success message if the deletion is successful, or an error if it fails.

Summary of API Endpoints in the API Layer

These API endpoints handle various product management operations such as retrieving, adding, updating, and deleting products using services and validators from the business logic layer. The usage of FluentValidation ensures that the incoming data is validated before processing, while the Results object simplifies generating responses.

Program.cs File in API Layer

This Program.cs file sets up and configures the services, middleware, and API endpoints for the eCommerce Products Microservice. Below is a breakdown of the configuration.

1. Builder Configuration

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

- The WebApplication.CreateBuilder(args) method initializes a new WebApplicationBuilder, setting up the application host.

2. Adding Services

```
// Add DAL and BLL services

builder.Services.AddDataAccessLayer(builder.Configuration);

builder.Services.AddBusinessLogicLayer();
```

- **Data Access Layer (DAL)** and **Business Logic Layer (BLL)** services are registered using extension methods AddDataAccessLayer and AddBusinessLogicLayer.

```
builder.Services.AddControllers();
```

- Adds the **MVC controllers** for handling incoming HTTP requests.

3. Fluent Validation Setup

```
// Fluent Validations

builder.Services.AddFluentValidationAutoValidation();
```

- Registers Fluent Validation for automatic validation of models in the API.

4. JSON Enum Converter

```
// Add model binder to read values from JSON to enum

builder.Services.ConfigureHttpJsonOptions(options => {

   options.SerializerOptions.Converters.Add(new JsonStringEnumConverter());

});
```

- Configures JSON serialization to support conversion between enum values and strings.

5. Swagger Configuration

```
// Add Swagger services

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();
```

- **Swagger** is added for API documentation and testing. The AddSwaggerGen method configures Swagger UI generation for the project.

6. CORS Configuration

```
// CORS

builder.Services.AddCors(options => {

  options.AddDefaultPolicy(builder => {

    builder.WithOrigins("http://localhost:4200")

      .AllowAnyMethod()

      .AllowAnyHeader();

  });

});
```

- **CORS (Cross-Origin Resource Sharing)** is configured to allow specific domains (in this case, http://localhost:4200, which might be for an Angular frontend) to make requests to the API.

7. Application Middleware Pipeline

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

- After configuring services, the application is built using builder.Build().

8. Middleware Setup

```
// Exception Handling Middleware

app.UseExceptionHandlingMiddleware();
```

- Adds custom **exception handling middleware** to handle any exceptions that occur during request processing.

```
app.UseRouting();
```

- Enables **routing** for the API.

9. Additional Middleware

```
// CORS
app.UseCors();
```

- Applies the configured **CORS policy**.

```
// Swagger
app.UseSwagger();
app.UseSwaggerUI();
```

- Enables **Swagger** for API documentation at runtime.

```
// HTTPS, Authentication, and Authorization
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
```

- Redirects HTTP requests to HTTPS, and configures **authentication** and **authorization** middleware.

10. API Endpoint Mapping

```
app.MapControllers();
app.MapProductAPIEndpoints();
```

- Maps **controller-based endpoints** and the custom-defined **product API endpoints** to the application.

11. Running the Application

```
app.Run();
```

- Starts the application and begins listening for incoming HTTP requests.

Summary of Program.cs File in API Layer

This Program.cs file sets up the core structure of the API, including service registration for the business logic and data access layers, CORS policy, Fluent Validation, Swagger, and middleware such as exception handling. It also maps the API endpoints for product management.