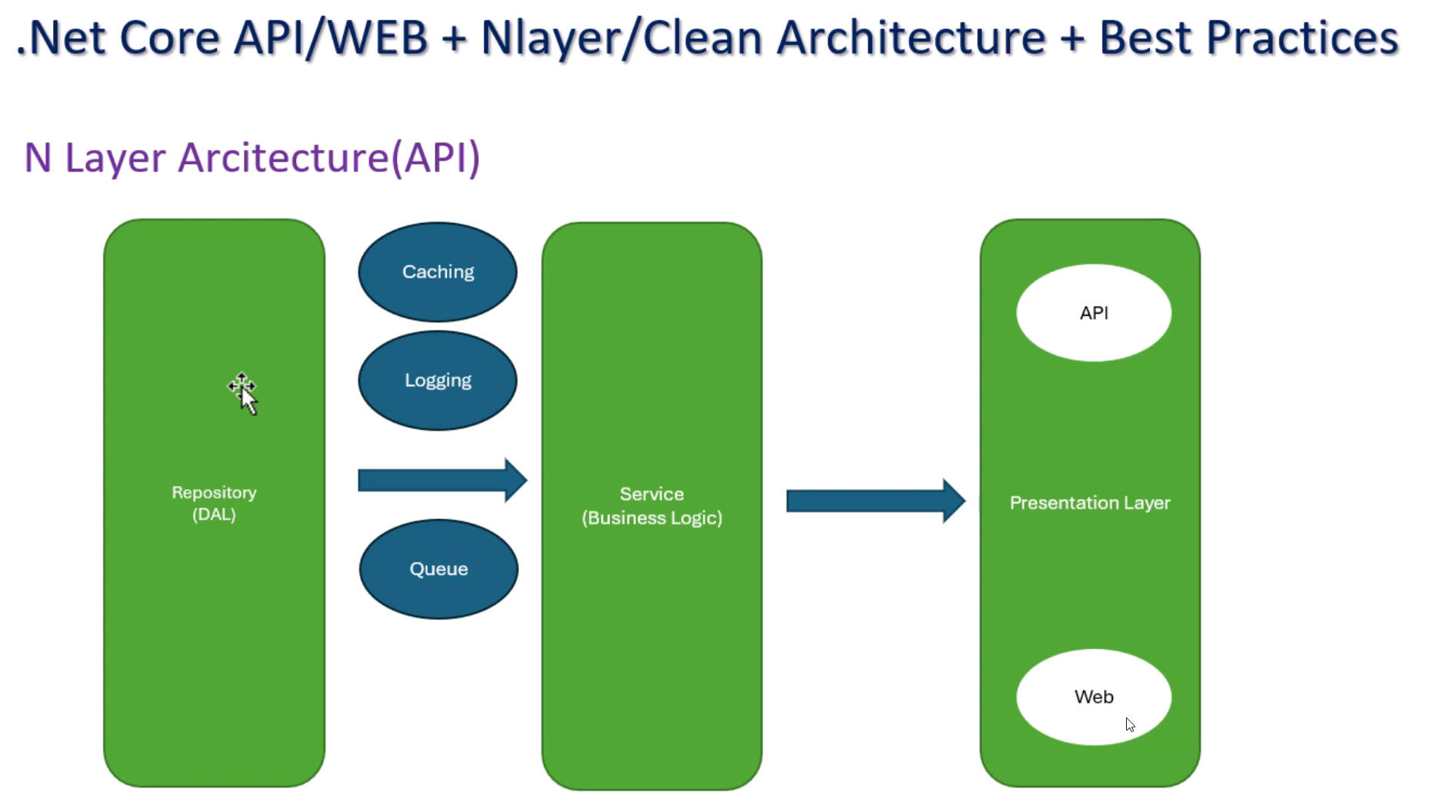
# Section 1: Giriş(New):

## 1. Kurs için Gerekli Olan IDE ve Programlar

# Section 2: N Layer Architecture ( New):

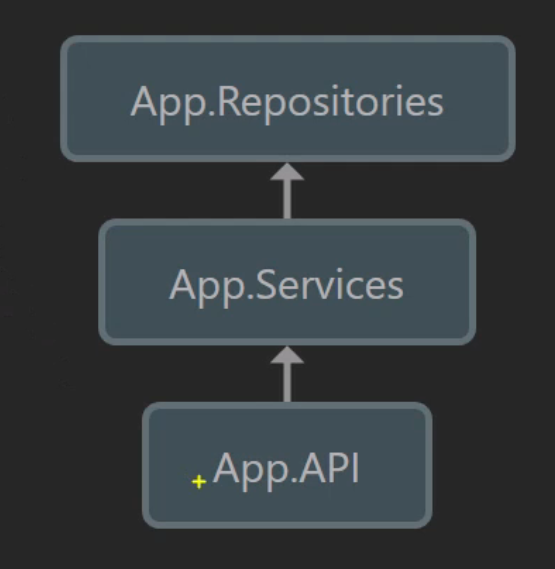
## 2. N Layer Architecture



* Repository katmanında try-catch olmayacak
* Repository katmanında DB ’den alınan data aynen Service katmanına aktarılacak.
* Service katmanı, Repository ‘den gelen data ile ilgili tüm işlemler bu katmanda yapılmalı. Try-catch burada yazılmalı.
* API, Service(Business Logic) katmanından aldığı datayı dış dünyada ulaştıran kısım. Controller’da Business logic olmamalı. Controller’da try-catch olmamalı.

## 3. Layers





.Net Framework 8.0 ile 3 tane proje oluşturuldu. App.API projesi Web API, diğerleri Class Library projeleri olacak şekilde. Aralarında ilişki yukarıdaki gibidir.

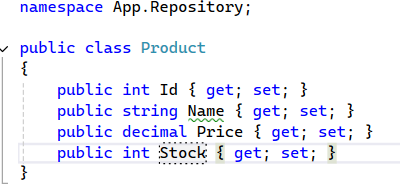
## 4. Repositories(DbContext - Entity)

* Aşağıdaki Nuget package ‘leri Repository projesine eklendi.

<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.8" />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.8">

* Null alanların yönetimi:



Name alanındaki uyarı, bu alan string olduğu için null olabilir. Bu durumda nasıl davranması gerektiğini belirt demek istiyor.

publicstring? **Name {** get; set; **}** ifadesi ile bu alanın boş geçilebilir olduğunu belirtmiş oluyorum.

public string Name { get; set; } = default!; ifadesi ile bu alanın boş geçilemez olduğunu belirtmiş oluyorum.

* Primary Constructor; aşağıdaki const. Kullanımı yerine

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

{

}

public DbSet<Product> Products { get; set; } = default!;

}

.Net 9.0 ile birlikte gelen primary const. Yapısı kullanılabilir. Primary Constructor ‘daki options Program.cs içerisinden ayarlanıyor.

public class AppDbContext(DbContextOptions<AppDbContext> options) : DbContext(options)

{

public DbSet<Product> Products { get; set; } = default!;

}

* EntityConfiguration’larını toplu uygulama.
  + Entity Config.’leri tek tek DbContext ‘e aşağıdaki gibi eklemek yerine

namespace App.Repository;

public class AppDbContext(DbContextOptions<AppDbContext> options) : DbContext(options)

{

public DbSet<Product> Products { get; set; } = default!;

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.ApplyConfiguration(new ProductConfiguration());

base.OnModelCreating(modelBuilder);

}

}

* + Aşağıdaki ifade ile Repository Assembly ‘sindeki IEntityTypeConfiguration interface’ini implemente eden tüm sınıfların config. Belirtmiş oluyoruz.

modelBuilder.ApplyConfigurationsFromAssembly(

Assembly.GetExecutingAssembly());

## 5. Repositories(Migration)

appsettings.json dan veri okumak için önceden aşağıdaki yöntemi kullanıyorduk. Controller’da Configuration’ ı include ediyorduk.

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

{

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

});

Bunun yerin Options pattern ‘i kullanıcaz.

public class ConnectionStringOption

{

public const string Key = "ConnectionStrings";

public string SqlServer { get; set; } = default!;

}

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

{

var connectionString = builder.Configuration.

GetSection(ConnectionStringOption.Key).Get<ConnectionStringOption>();

options.UseSqlServer(connectionString!.SqlServer);

});

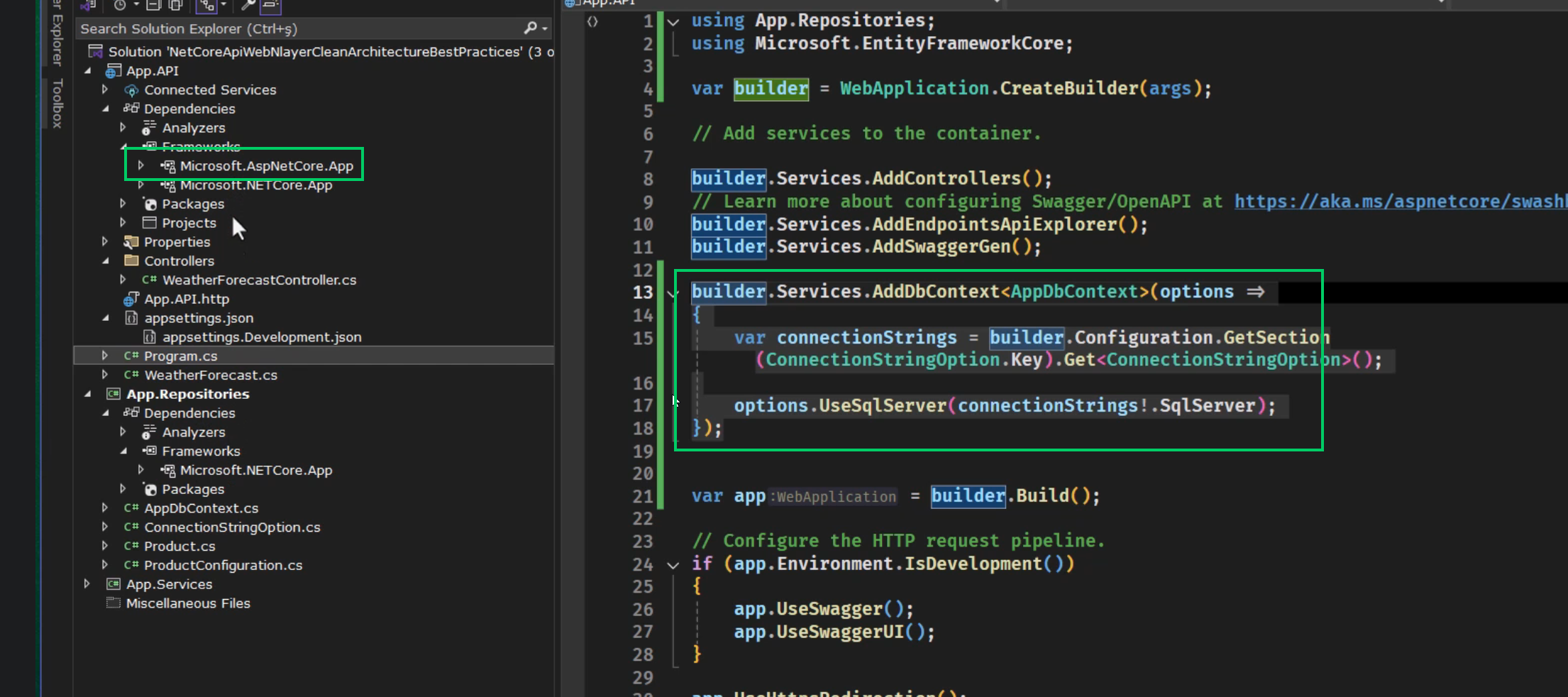
connectionStrings.SqlServer ‘da compiler uyarı veriyor.

connectionStrings!.SqlServer ile compiler ‘a böyle bir data var, bu alan null gelmiyecek diyoruz. ! işareti ile compiler in verdiği warning i uzaklaştırıyoruz sadece, başka bir manası yok.

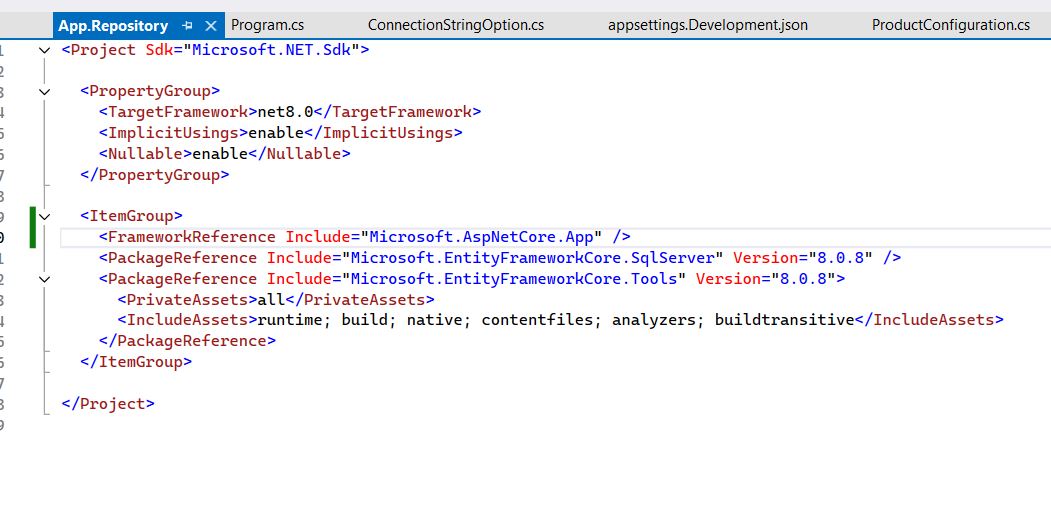
## 6. Repositories(Migration)-2

AddDbContext ile API’deki DBContext config. ‘unu Repository ‘ye taşımak istiyoruz. Böylece API ‘deki Services kısmında yukarıdaki kod bloğu olmayacak ve ilgili kısım sadece Repository ‘de olacak.

Bunun için Services ‘i Repository ‘de kullanmak için, Microsoft.AspNetCore.App Framework ‘unu Repository’ye taşımam gerekli.



Repository projesini aşağıdaki gibi değiştirdik.



Şimdi Repository ‘de bu kısmı Extension Method olarak yazarak, API ‘de kullanabiliriz.

namespace App.Repository.Extensions

{

public static class RepositoryExtensions

{

public static IServiceCollection AddRepositories(this IServiceCollection services, IConfiguration configuration)

{

services.AddDbContext<AppDbContext>(options =>

{

var connectionStrings = configuration.GetSection(ConnectionStringOption.Key).Get<ConnectionStringOption>();

options.UseSqlServer(connectionStrings!.SqlServer, sqlServerOptionsAction =>

{

// AppDbContext 'in yeri değişebilir. Migration 'ların Repository 'de kalması için

// RepositoryAssembly adında bir struct yazdık ve Assembly bilgisini alırken bunu kullandık

// sqlServerOptionsAction.MigrationsAssembly(typeof(AppDbContext).Assembly.FullName);

sqlServerOptionsAction.MigrationsAssembly(typeof(RepositoryAssembly).Assembly.FullName);

});

});

return services;

}

}

}

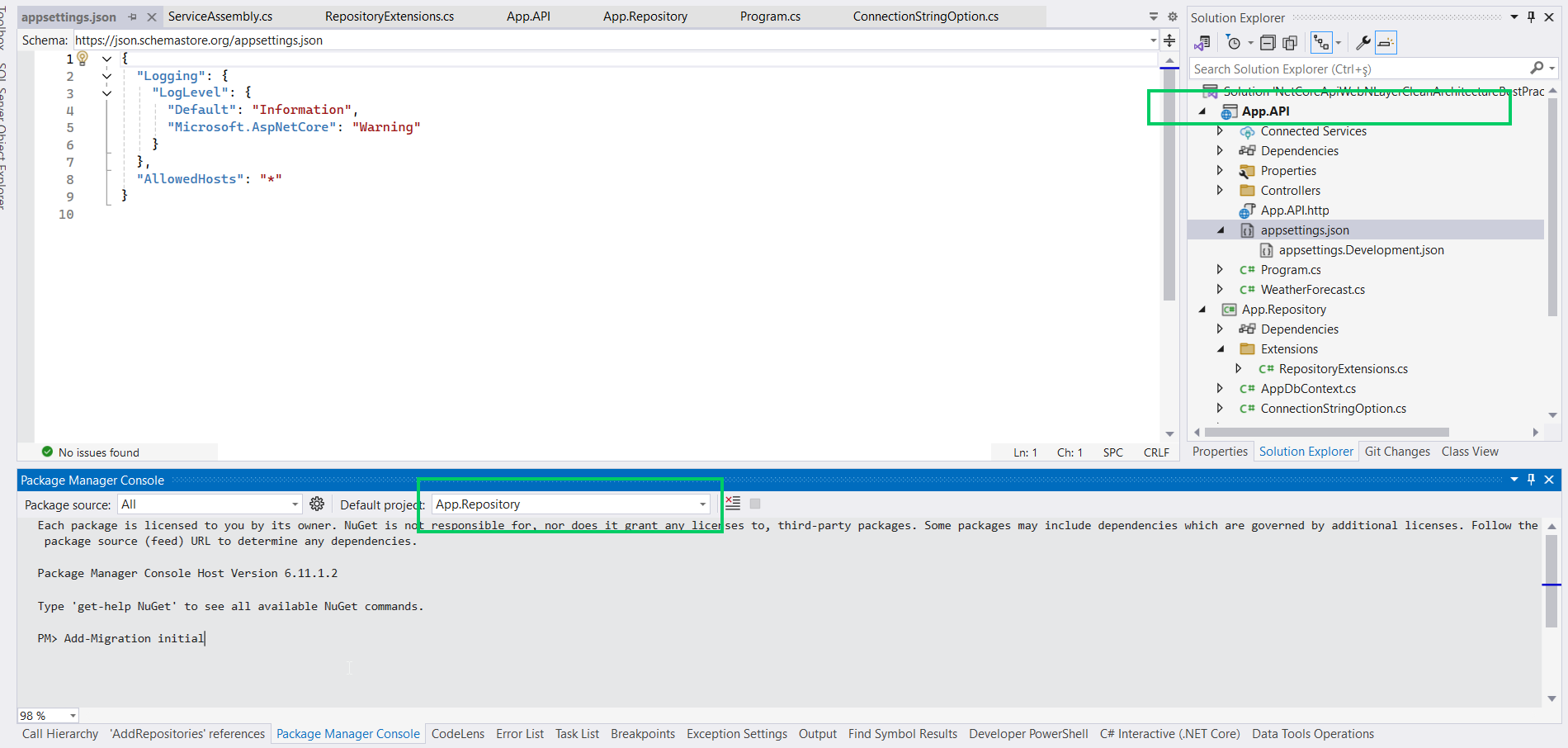
Migration kodlarının repository projesinde oluşması için bir options tanımlandı. Bu options ‘da sabit bir class’ın olduğu proje assembly ‘sini kullanmak için RepositoryAssembly isimli bir struct, class vb. De olaiblirdi tanımlandı.

RepositoryAssembly strcut ‘i aşağıdaki gibidir:

namespace App.Repository;

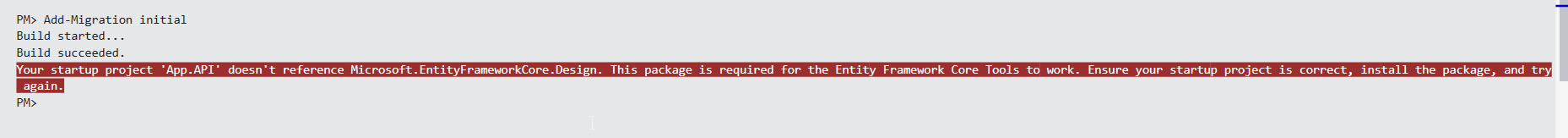
public struct RepositoryAssembly;

API ile DbContext ilişkilendirmek için sadece aşağıdaki extension metodu kullanıyoruz.



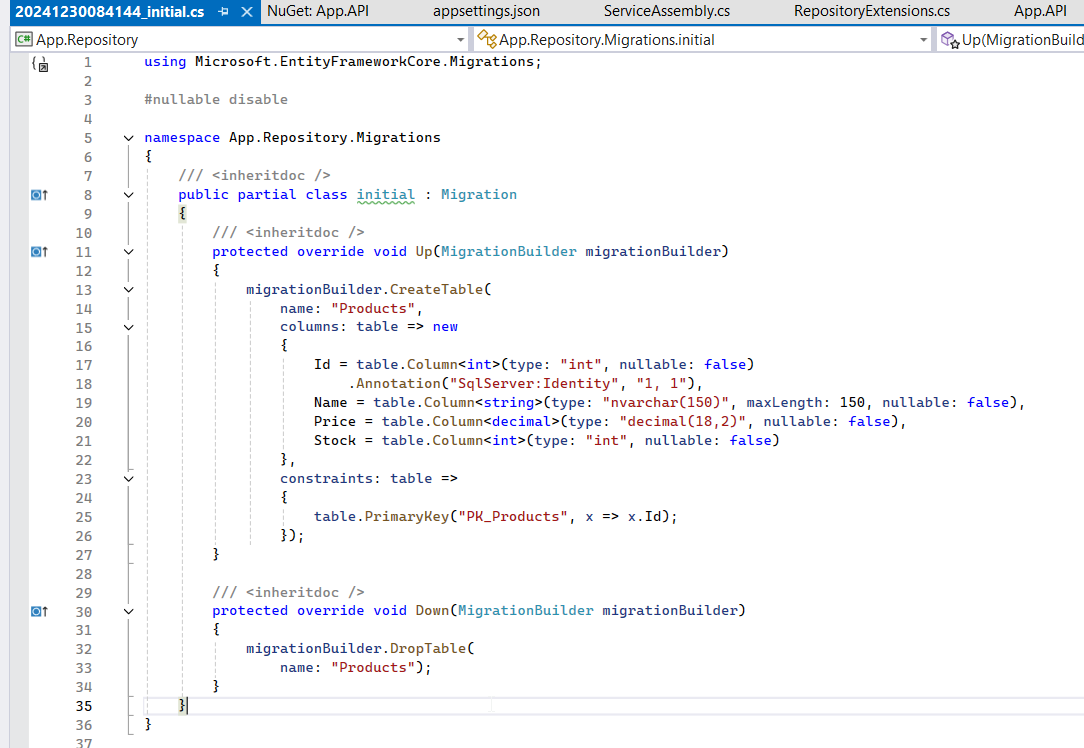
Migration denemesi yapmak için projeyi derledik, startup projesini API yaptık ve Package Manager Console ‘da App.Repository projesini seçtik.

Add-Migration initial communu çalıştırdık.



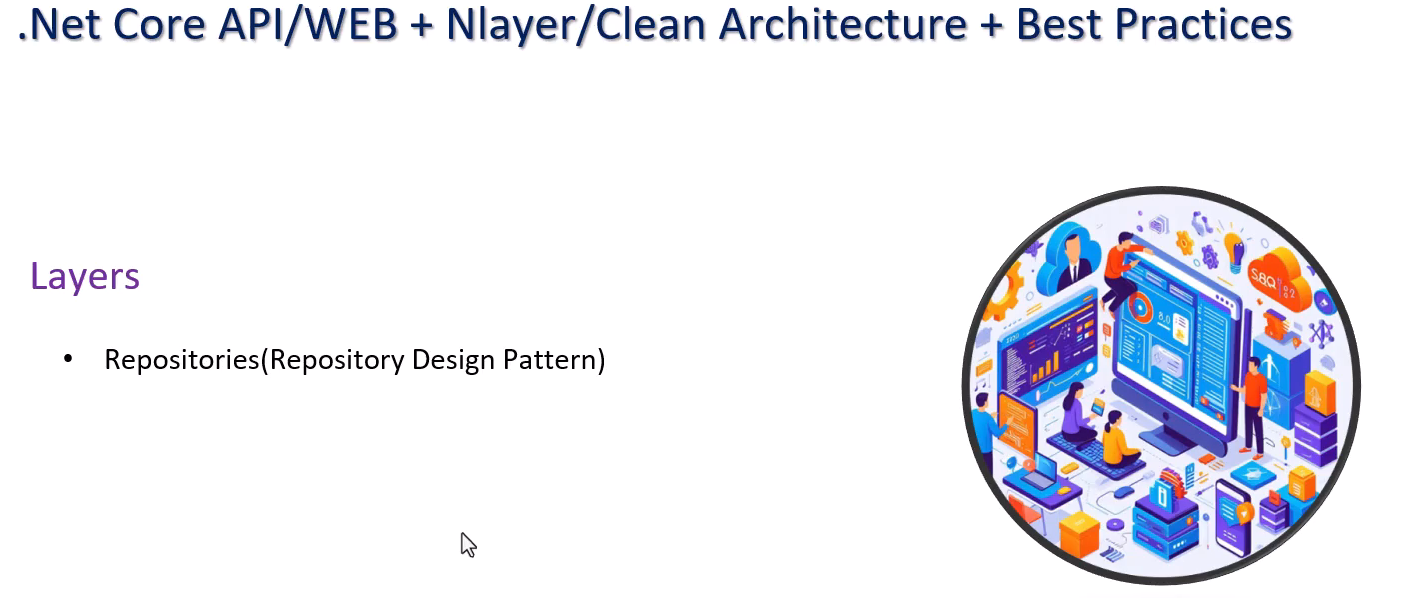
Microsoft.EntityFrameworkCore.Design nuget package ‘İni API projesine ekledim.

Tekrar migration komutunu verdik ve migration oluştu.



Update-Database ile tabloları Sql Server DB’sinde oluşturuyoruz.

## 7. Repository Design Pattern



Repository Design Pattern kullanıp, generic bir yapı inşa edilecek. Generic class ‘ları çok fazla metod ekleyerek şişirmeye gerek yok. Örneğin, bir projemizde 60 entity var ve bazılarında custom query ‘ler yazacak isek Generic Repository ‘ye bu custom metodları ekleyerek onu büyütmek doğru değil. İlgili Entity : IRepository ‘den miras alarak extend etmek daha doğru.

using System.Linq.Expressions;

namespace App.Repository;

internal interface IGenericRepository<T> where T : class

{

IQueryable<T> GetAll();

IQueryable<T> Where(Expression<Func<T, bool>> predicate);

ValueTask<T?> GetByIdAsync(int id);

ValueTask AddAsync(T entity);

void Update(T entity);

void Delete(T entity);

}

using Microsoft.EntityFrameworkCore;

using System.Linq.Expressions;

namespace App.Repository;

public class GenericRepository<T>(AppDbContext context) : IGenericRepository<T> where T : class

{

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

public IQueryable<T> GetAll() => \_dbSet.AsQueryable().AsNoTracking();

public IQueryable<T> Where(Expression<Func<T, bool>> predicate) => \_dbSet.Where(predicate).AsQueryable().AsNoTracking();

public ValueTask<T?> GetByIdAsync(int id) => \_dbSet.FindAsync(id);

public async ValueTask AddAsync(T entity) => await \_dbSet.AddAsync(entity);

public void Update(T entity) => \_dbSet.Update(entity);

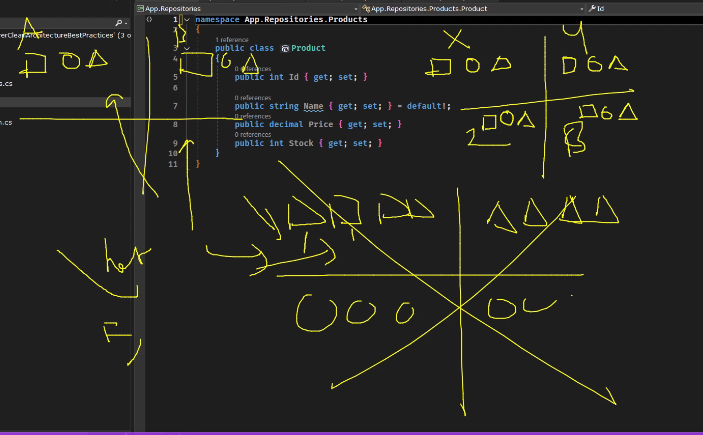
public void Delete(T entity) => \_dbSet.Remove(entity);

}

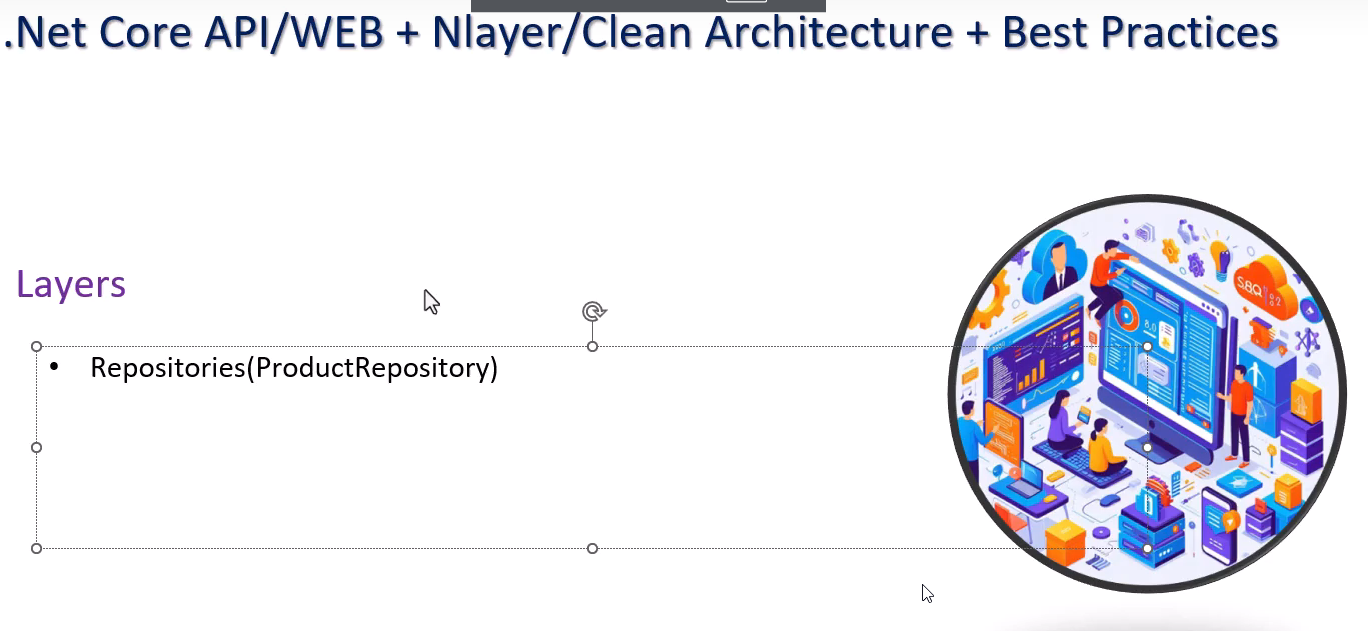
readonly DbSet<T> \_dbSet ile dbSet

* Readonly olarak tanımlanması, nesnenin dışarıdan değiştirilemeyeceği anlamına gelir. Bu, repository'nin veri tutarlılığını korur; yani veri kaynaklarını yöneten sınıflarınızda kontrol kaybını önler.
* **readonly** kullanarak, **DbSet**'in yanlışlıkla değişmesini önleriz. Bu tür durumlar, veri tutarsızlıklarına ve bakım zorluklarına yol açabilir.
* **readonly** alanlar, bellekte daha az yük tutarak performansa katkıda bulunabilir.

Proje bazlı ayrım yerine(Entity, Configuration vb.) model bazlı ayrımın büyük projelerde daha takip edilebilir olduğunu belirtti. Şöyle bir örnek verdi, X,Y,Z,W isimli ressamların, A, B, C, D isimli eserleri olsun. Bir müzeye gittiğinizde 4 ayrı odada eserler sergilenecek olsun, bu durumda eserlere göre kategorilernmiş odalar mi istersiniz, ressamlara göre kategorilenmiş odalarmı. Bu yüzden solution ‘da Entity için ayrı proje, Mapping ‘ler için ayrı projeler yerine ilişkili modelleri bir klasör altında toplamak daha kullanışlı olduğunu belirtti. Bu yüzden Repository altında Products klasörü oluşturup Product ve ProductConfiguration classlarını bunun altına taşıyarak kategorize etmenin daha kullanışlı olduğunu belirtti.



## 8. Product Repository



ProductService ‘inde db işlemlerini içerecek kodlar olmasın. BU kodlar ProductRepository ‘de olsun. Böylece Service daha yalın ve sadece business’ları içerir. Repository ise DB. işlemlerini içerir.

public Task<List<Product>> GetTopPriceProductsAsync(int count)

{

return context.Products.OrderByDescending(p => p.Price).Take(count).ToListAsync();

}

Yukarıdaki metodda async ve await kullanmadık, çünkü gerek yok. Sadece data döndüğü için Compile edilince yukarıdaki hale dönüştürülüyor.

public async Task<List<Product>> GetTopPriceProductsAsync(int count)

{

return await context.Products.OrderByDescending(p => p.Price).Take(count).ToListAsync();

}

IproductRepository interface i oluşturuldu. IgenericRepository miras alıyor. Böylece IgenericRepository tüm metodlar bu interface de de olmalı.

namespace App.Repository.Products;

public interface IProductRepository : IGenericRepository<Product>

{

public Task<List<Product>> GetTopPriceProductsAsync(int count);

}

ProductRepository constructor ‘ında Context belirtiliyor ve GenericRepository miras aldığı için IgenericRepository interface’indeki metodlar implemente edilmiş oluyor. Sadece IproductRepository ‘deki metod implemente edilecek.

using Microsoft.EntityFrameworkCore;

namespace App.Repository.Products;

internal class ProductRepository(AppDbContext context) : GenericRepository<Product>(context), IProductRepository

{

public Task<List<Product>> GetTopPriceProductsAsync(int count)

{

return Context.Products.OrderByDescending(p => p.Price).Take(count).ToListAsync();

}

}

GenericRepository ‘ye protected AppDbContext Context = context; eklendi. Böylece miras alan sınıflarda Context ile sorgular yazılacak.

public class GenericRepository<T>(AppDbContext context) : IGenericRepository<T> where T : class

{

protected AppDbContext Context = context;

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

public IQueryable<T> GetAll() => \_dbSet.AsQueryable().AsNoTracking();

public IQueryable<T> Where(Expression<Func<T, bool>> predicate) => \_dbSet.Where(predicate).AsQueryable().AsNoTracking();

public ValueTask<T?> GetByIdAsync(int id) => \_dbSet.FindAsync(id);

public async ValueTask AddAsync(T entity) => await \_dbSet.AddAsync(entity);

public void Update(T entity) => \_dbSet.Update(entity);

public void Delete(T entity) => \_dbSet.Remove(entity);

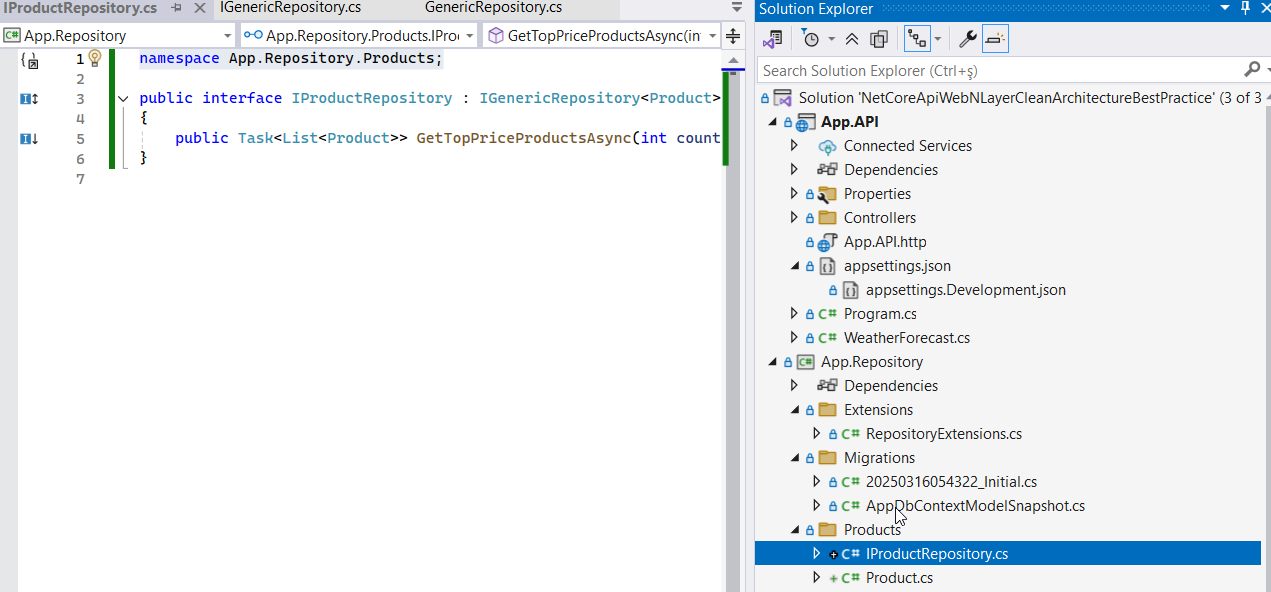
}

## 9. Product Repository DI Container



Proje dosya yapısındaki isimlendirme ile namespace isimleri uygun olmalı. App.Repository.Products dizini altındaki tüm dosyaların namespace i namespace App.Repository.Products; olmalı.

Daha anlaşılabilir ve kolay takip sağlamak için.



Repository class ‘larının yaşam ömrünü, RepositoryExtensions içerisinde aşağıdaki gibi belirtiyoruz.

DbContext ve EF kullanıldığı için AddTransient ya da AddSingleton kullanılmaz. Yaşam döngüsü sonlandığı anda bu nesneler de Dispose edilsin demiş oluyoruz. DbCOntext ‘in Yaşam Döngüsü Scoped olmalıdır. Request ile nesne oluşur, Response döndüğünde DbContext ‘lerde dispose edilir. O yüzden bu repository’lerde AddScoped olmalıdır. EFCore için geçerli bir yaklaşım. Başka bir veri erişim framework ‘u singleton ya da transient önerebilir. DbContext, Transient ya da Singleton olmaz. Veri bütünlüğü bozulur.

services.AddScoped<IProductRepository, ProductRepository>();

services.AddScoped(typeof(IGenericRepository<>),typeof(GenericRepository<>));

IGenericRepository gibi IGenericService yapmak çok sağlıklı değil diyor!

Eğer, BaseService yapıp ortak metodlar kullanılmayacaksa.

App.Service projesinde de klasör yapısı oluşturuldu. İlgili class’lar ilgili Klasörler altında yer alacak şekilde. Aynı şekilde Extensions klasörü oluşturuldu, Service ile ilgili extensions’ları yazmak için. App.Service project dosyası edit edilerek AspNetCore framework ‘ü de projeye eklendi. Service ile ilgili startup confg. Buradan yapılması için.

<ItemGroup>

<FrameworkReference Include="Microsoft.AspNetCore.App" />

<ProjectReference Include="..\App.Repository\App.Repository.csproj" />

</ItemGroup>

Program.cs ‘de Services class ları için yazdığımız Extension ‘i çağırıyoruz. Böylece Services class ‘larının nasıl türetileçeğini belirtmiş oluyoruz.

builder.Services.AddRepositories(builder.Configuration).AddServices(builder.Configuration);

public static class ServiceExtensions

{

public static IServiceCollection AddServices(this IServiceCollection services, IConfiguration configuration)

{

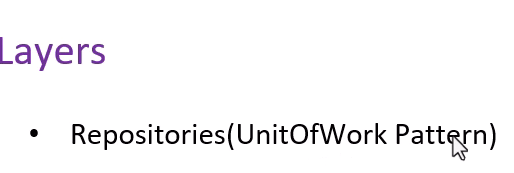
services.AddScoped<IProductService, ProductService>();

return services;

}

}

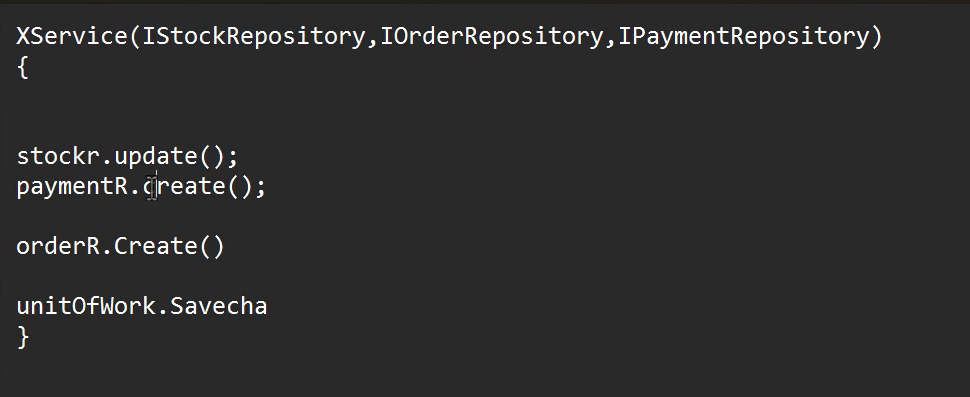
## 10. UnitOfWork



UnitOfWork pattern transaction’ın bütünlüğünü saplayan bir pattern ‘dir.

Repository metodlarında hiç SaveChanges kullanmadık. Bunu UnitOfWork yapacak.

Aşağıdaki gibi üç işlemin olduğu bir senaryoda tüm işlemlerin commit ya da rollback olması gerekli. UnitOfWork kullanılmaz ise işlem yarım gerçekleşebilir.



namespace App.Repository;

public interface IUnitOfWork

{

Task<int> SaveChangesAsync();

}

public class UnitOfWork(AppDbContext context) : IUnitOfWork

{

public Task<int> SaveChangesAsync() => context.SaveChangesAsync();

}

Alttaki 3 SaveChangesAsync metodu da aynı. İlki Lambda expression. 2. Async await siz hali. Eğer sadece savechanges ve return yapılacak ise async await kullanmaya gerek yok. 3. Async await kullanılmış hali.

public Task<int> SaveChangesAsync() => context.SaveChangesAsync();

public Task<int> SaveChangesAsync()

{

return context.SaveChangesAsync();

}

public async Task<int> SaveChangesAsync()

{

return await context.SaveChangesAsync();

}

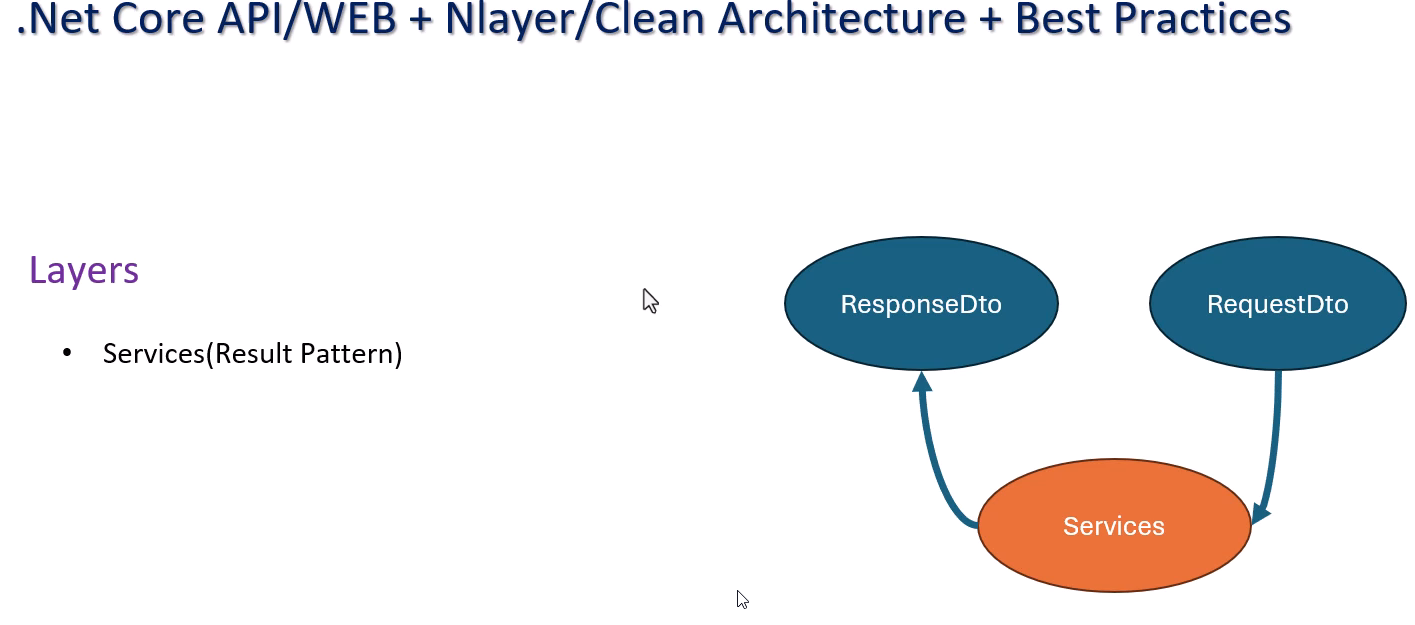
services.AddScoped<IUnitOfWork, UnitOfWork>();

Business Logic ‘te Veritabanını gitmiyor isen Singleton olabilir. Ama veritabanı gidiyorsan Scoped olmalı.

Controller ‘da kesinlikle Repository class’ları kullanılmayacak. Controller, Service classlarını kullanmalı her durumda. Service, repository ‘ye gitmeli.

Controller’da BusinessLogic olmayacak.

## 11. Result Pattern



Service katmanı, Dto alır ve geriye Dto döner. ResponseDto, başarılı durumda datayı taşımalı, başarısız durumda da hatayı taşımalı.

Ortak bir model dönecek classımızı aşağıdaki gibi implemente ettik:

namespace App.Service;

public class ServiceResult<T>

{

/// <summary>

/// Başarılı olduğunda Data dolucak

/// </summary>

public T? Data { get; set; }

/// <summary>

/// Başarısız olduğunda hata mesajı dönecek

/// </summary>

public List<string>? ErrorMessage { get; set; }

/// <summary>

/// Lambda expression

/// </summary>

public bool IsSuccess => ErrorMessage == null || ErrorMessage.Count == 0;

//public bool IsSuccess

//{

// get

// {

// return ErrorMessage == null || ErrorMessage.Count == 0;

// }

//}

public bool IsFail => !IsSuccess;

}

IsFail property ‘si önemli aslında. Projenin herinde if ile IsSuccess değil ise yazmak yerine bunu kullanmak çok daha mantıklı.

ProductService ‘ten artık direkt entity yerine bu modeli dönücez:

Eski hali;

public class ProductService(IProductRepository productRepository) : IProductService

{

public Task<List<Product>> GetTopPriceProductsAsync(int count) => productRepository.GetTopPriceProductsAsync(count);

}

Örnek bir Service çağrısı ve ServiceResult geri dönüşü aşağıdaki gibi implemente edilebilir.

public async Task<ServiceResult<Product>> GetProductByIdAsync(int id)

{

var product = await productRepository.GetByIdAsync(id);

if (product is null)

{

return new ServiceResult<Product>

{

ErrorMessage = new List<string> { "Product not found!" }

};

}

return new ServiceResult<Product>

{

Data = product

};

}

ServiceResult’ı projenin her yerinde new’lemek yerine ServiceResult ‘ta Extension Metodlar yazarak bu new’leme işleminden kurtulalaım ve yeni ServiceResult oluşturma işlemini merkezi bir yerde yapmış olalım.

ServiceResult’a static Success ve Fail metodları eklendi. Aynı zamanda public HttpStatusCode Status { get; set; } eklendi. İşle sonucunda Http durumunu set etmek için.

public class ServiceResult<T>

{

/// <summary>

/// Başarılı olduğunda Data dolucak

/// </summary>

public T? Data { get; set; }

/// <summary>

/// Başarısız olduğunda hata mesajı dönecek

/// </summary>

public List<string>? ErrorMessage { get; set; }

public bool IsSuccess => ErrorMessage == null || ErrorMessage.Count == 0;

//public bool IsSuccess

//{

// get

// {

// return ErrorMessage == null || ErrorMessage.Count == 0;

// }

//}

public bool IsFail => !IsSuccess;

public HttpStatusCode Status { get; set; }

/// static factory methods

/// new lemeyi kontrol altına almış olduk

public static ServiceResult<T> Success(T data, HttpStatusCode status = HttpStatusCode.OK)

{

return new ServiceResult<T>()

{

Data = data,

Status = status,

};

}

public static ServiceResult<T> Fail(List<string> errorMessages, HttpStatusCode status = HttpStatusCode.BadRequest)

{

return new ServiceResult<T>()

{

ErrorMessage = errorMessages,

Status = status

};

}

public static ServiceResult<T> Fail(string errorMessage, HttpStatusCode status = HttpStatusCode.BadRequest)

{

return new ServiceResult<T>()

{

// ErrorMessage = new List<string> { errorMessage }

ErrorMessage = [errorMessage],

Status = status

};

}

}



## 12. Result Pattern-2

## 13. Controllers

## 14. Endpoint names

## 15. Pagination Endpoint

## 16. CreateEndpoint

## 17. UpdateStock Endpoint

## 18. Route Constraint

# Section 3: Validation (New):

## 19. Model Validation-1

## 20. Model Validation -2

## 21. Model Validation -3

## 22. Model Validation 4

# Section 4: Mapping(New):

## 23. AutoMapper

# Section 5: Exception Handler(New):

## 24. ExceptionHandler

# Section 6: API Geliştirme(New):

## 25. Update Product Validator

## 26. Product Add/Update Mapping

## 27. Category Entity/Repository

## 28. Category Service

## 29. Product Service(Güncelleme)

## 30. SaveChanges Interceptor

# Section 7: Best Practices(New):

## 31. Http Method Types

## 32. Endpoint Names

## 33. Http Status Codes

## 34. Program.cs

## 35. Action Method

## 36. Exception

## 37. Return Type

## 38. Folder Structure

## 39. Filters

## 40. Visual Studio Built-int API Tester

# Section 8: Clean Architecture (New):

## 41. Principles

## 42. Dependency Inversion/Inversion Of Control

## 43. Single Responsibility Principle (SRP)

## 44. Persistence Ignorance/Separation Of Concern

## 45. Clean Architecture

## 46. Clean Architecture Proje Structure

## 47. Domain

## 48. Application

## 49. Instrastructure

## 50. API

# Section 9: Add Feature(Caching) (New):

## 51. Caching

## 52. Pragram.cs Refactoring

# Section 10: Service Bus ( RabbitMQ) (New):

## 53. Service Bus - 1

## 54. Service Bus - 2

# Section 11: Giriş(Eski):

## 55. Giriş

# Section 12: Asp.Net Core Best Practices(Eski):

## 56. Http Method Tipi Seçimi

## 57. Doğru Api/Endpoint Yapısı

## 58. Doğru Http Cevap Durum Kodları

## 59. Doğru Endpoint Url Yapısı

## 60. Request içerisinde aynı property’i almaktan kaçın

## 61. Asp.net Core uygulamanın startup.cs Dosyasını mümkün olduğunca sade bırak

## 62. Uygulamanızı mümkün oldukça küçük parçalara böl

## 63. Controller sınıflarınız mümkün oldukça temiz tutun. Business kodu bulundurmayın.

## 64. Action methodlarınızı mümkün oldukça temiz tutun. Business kodu bulundurmayın.

## 65. Hataları global olarak ele alın. Action methodlar içerisinde try catch blokları

## 66. Tekrar eden kodlardan kaçınmak için filter kullan

## 67. Action methodlardan direk olarak model sınıflarınızı dönmeyin

Section 13: Asp.Net Core API + Katmanlı Mimari(Eski):

68. N-Layer Proje Yapısı

69. Projelerimizin oluşturulması

Section 14: Core Layer(Eski):

70. Entity’lerimizin oluşturulması

71. .Net6 nullable

72. IGenericRepository

73. IService

74. IUnitOfWork

Section 15: Repository Layer(Eski):

75. AppDbContext

76. Entity Configurations

77. Seed Data

78. GenericRepository

79. UnitOfWork

80. Migrations

Section 16: Service Layer(Eski):

81. Service

82. DTOs

83. AutoMapper ( entity <> Dto)

84. CustomResponseDto

Section 17: Asp.Net Core API App(Eski):

85. ProductController

86. ProductController Test

87. ProductsWithCategory Endpoint

88. GetCategoryByIdWithProducts Endpoint

89. IProductService

90. FluentValidation Library

91. FluentValidation Library-2

92. Global Exception Handler

93. Service katmanında hata fırlatmak

94. NotFoundFilter

95. AutoFac

96. CachingService-1

97. CachingService-2

98. CachingService-3

Section 18: Asp.Net Core MVC App(Eski):

99. Giriş/ProductController-1

100. ProductController-2

101. ProductsController Save-1

102. ProductsController Save-2

103. FluentValidation

104. ProductsController Update-1

105. ProductsController Update-2

106. ProductsController Delete

107. NotFoundFilter

108. CreatedDate/UpdatedDate

Section 19: Asp.Net Core API-MVC(Eski):

109. Giriş

110. ProductApiService/CategoryApiService

111. ProductApiService Methods

112. CategoryApiService Methods

113. productsController Update / Test

Section 20: Net 7(Eski):

114. Ek Dersler

115. Net 7 Upgrade

Section 21: Generic Service Layer-DTO Refactoring Net 7(Eski):

116. IServiceWithDto interface

117. ServiceWithDto class

118. ProductServiceWithDto class

119. ProductWithDtoController class

120. Override SaveChanges()

121. CatagoryWithDtoController