BERKAY AKAR

  Bilgisayar Mühendisi

DAPPER TEKNOLOJİSİ

Contents

[DAPPER 2](#_Toc94975788)

[DAPPER KULLANIMI 2](#_Toc94975789)

[Connected Mimaride ( ADO.NET – DAPPER ): 2](#_Toc94975790)

[EXECUTE INSERT: 4](#_Toc94975791)

[EXECUTE SCALAR 5](#_Toc94975792)

[DYNAMIC & VAR 7](#_Toc94975793)

# DAPPER

Dapper StackOverflow Ekibi Tarafından Geliştirilmiş Bir Micro ORM ARACIDIR. Diğer ORM’leri baz alarak oldukça Hızlıdır. Entity framework gibi Code FIRST Özelliği yoktur. Bu sebeple var olan bir veritabanları üzerinde geliştirmeye aday bir ORM aracıdır. Halihazırda oluşturulmuş bir database e ihtiyaç vardır.

tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu

Şekil 1 ORM ARAÇLARI TEPKİ SÜRELERİ

Bu sebeple Dapper = **MICRO ORM KRALI** olarak adlandırılır.

# DAPPER KULLANIMI

Daha öncesinde Dapper’i anlatırken bir Veri tabanını olması gerektiğinden bahsetmiştik. Bu sebeple önce süreçleri işleyebilmek adına bir veritabanı oluşturuyoruz.

CREATE DATABASE DapperDb Veritabanı oluşturma

USE DapperDb Veritabanı Kullanma (varsayılan olarak Master veritabanı gelir)

create table Products(

Id int identity(1,1),

Name nvarchar(50),

Price decimal(18,3),

Stock int,

primary key(Id)

)

## Connected Mimaride ( ADO.NET – DAPPER ):

Yapının Daha iyi Anlaşılabilmesi için kodları karşılaştırmak gerekir. Basit bir insert işlemini karşılaştıralım.

* İş sürecine Sırası ile bakmak gerekirse

using System.Data.SqlClient; // İLK OLARAK KÜTÜPHANEMİZİ ÇEKTİK. (ADO.net için)

* Veri Tabanı Bağlantısı için Nesne Oluşturumu

SqlConnection conn = new SqlConnection(); // Connectipn atabilmrmiz için oluşturuldu Beraberinde overload li yordamlar kullanılarak çeşitli parametreler ile de obje oluşturabilmemiz mümkündü.

* Bağlanılacak olan Veritabanı stringi

conn.ConnectionString = "server=BERKAY\_MONSTER; database=DapperDb ; integrated security = true";

* Belirtilen nesnenin Veritabanına erişimi için Açılması

conn.Open();

* Çalıştırışacak komutun belirlenmesi

SqlCommand cmd = conn.CreateCommand();

* Sorgunun oluşturulması

cmd.CommandText = "insert into Products values ('samsung',4000,30)";

* Oluşturulan iş sürecinin veritabanına çeşitli parametreler ile aktarımı :

cmd.ExecuteNonQuery(); -- Kaç Satır Etkilenmiştir Cevabını Döndürmekte

* Çaliştirilan sorgunun Kapatılması

conn.Close();

Göüzktüğü üzere toplamda 8 adımdan oluşan bu içerisinde overloadlı yordamlar kullanılarak dahada kıslatılabilirdi. Ama temelinde iş süreçleri aynı bulunmaktadır. DAPPER kütüphanesi ile işlemleri tekrarlayalım ve kendisine özel tekrarlayalım

* İlk Aşamada kütüphanesini Dahil etmemiz gerekecektir. Dapper SQLCLİENT kütüphanesinden türetilme bir kütüphanedir.

using Dapper;

* Kodlar içeriksel ve işlevsel olarak benzerlik gösterdiği için direk açıklama ihtiyacı duymadan direk ekledim.

var connection = new SqlConnection();

connection.ConnectionString = "server=BERKAY\_MONSTER; database=DapperDb ; integrated security = true";

connection.Execute("insert into Products values('Dapper Geldi Hoş Geldi ',20,1000)");

Console.WriteLine("Mission Completed");

Bu şekilde tanımlanmıştır.

### EXECUTE INSERT:

Bizlere özünde işlenen satır sayısı bilgisi gönderir. SqlDataClient kütüphanesindeki executenonquery() gibi

var connection = new SqlConnection();

connection.ConnectionString = "Server=(localdb)\\MSSQLLocalDB; database=DapperDb; integrated security=true;";

connection.Execute("insert into Products values('Klavye',20,1000)");

Console.WriteLine("Mission Completed");

#### EXECUTE PARAMETRELİ INSERT

connection.Execute("insert into Products values(@name,@price,@stock)", new

{

name = "Laptop",

price = 4500,

stock = 50

}); //obje

Tekli objeler için bu şekilde bir atama yapmak özünde mümkündür. Fakat bierden çok dizi tipindeki veriyi aktarmaya kalkacaksak eğer :

connection.Execute("insert into Products values(@name,@price,@stock)", new[]

{

new {name="Moitör", price=4500, stock=30},

new {name="Mouse", price=500, stock=25}

});

İçerisine Veri atarken @name , @price gibi değişkenler veritabanındaki sutunlar ile aynı olmak zorunda değildir. Ama içerideki oluşturulan dizilerin elemanları string içerisinde verilen yapılar ile karşılayabilmesi açısından aynı olması gerekmektedir.

Parametreli ve çoklu Insert işlemi

#### EXECUTE UPDATE

connection.Execute("update Products Set Name='Kitap', Stock=500 where Id = 2");

Tek bir kaydı değiştirmek istiyorsanız bir obje rotasyonu birden fazla kaydı güncellemek istiyorsanız bir dizi rotasyonu geçiyoruz

connection.Execute("update Products Set Name=@name, Stock=@stock, Price=@price where Id = @id", new[]

{

new {name = "Kitap", price = 50, stock = 2000, id = 1},

new {name = "Kitap", price = 50, stock = 2000, id = 2},

new {name = "Kitap", price = 50, stock = 2000, id = 3},

new {name = "Kitap", price = 50, stock = 2000, id = 4},

new {name = "Kitap", price = 50, stock = 2000, id = 5},

});

İnsert işlemi ile benzerlik göstermektedir.

#### EXECUTE DELETE

connection.Execute("DELETE FROM Products where Id=@id", new[]

{ new { id = 1 }, new { id = 2 } });

connection.Execute("DELETE FROM Products where Id=@id", new[]

{ new { id = 24 }, new { id = 23 }, new { id = 22 }, new { id = 21 } });

connection.Execute("DELETE FROM Products Where Id = @id", new { id = 18 });

connection.Execute("DELETE FROM Products Where Id = 17");

### EXECUTE SCALAR

Esasinda execute gibi tüm sorguyu çalıştırır. İlk Satır İlk sütundaki değerleri getirtir önümüze.

var value = connection.ExecuteScalar("select \* from Products");

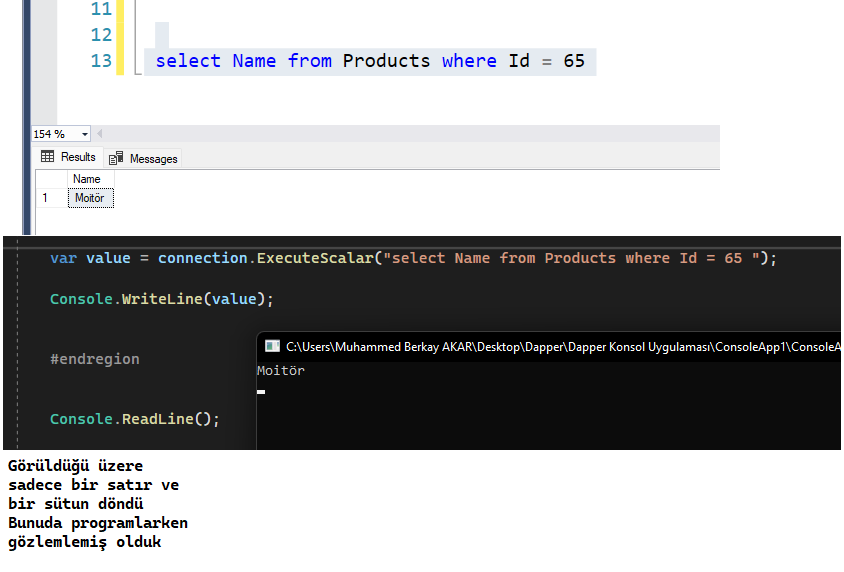
İlk değeri getirir. İlk değerimiz Id = 64 olduğu için ekrana Gelen değer: 64 yazar

Her zaman ilk satırdaki ilk kolondaki değeri getirir

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bir başka örnek daha yapalım.



Görüldüğü üzere sadece bir satır ve sütun bilgisi çekmektedir.

var value = connection.ExecuteScalar<string>("select name from Products");

İlgili değerin string olduğundan emin iseniz bu şekilde bir harekette bulunabilirsiniz

var value = connection.ExecuteScalar<string>("select name from Products where id=@id", new { id = 5});

Bu şekilde de where koşuluna gerekli parametreleri yollayarak iş yapabilmekteyiz.

Beraberinde execute scalar ef deki find ile benzer özellik göstermektedir. İkisinde de sadece bir satır ve bir kolon dönmektedir. Daha fazla bilgi (tüm satır bilgileri için) first or default kullanılması ve bunun üzerinden bir linq yapılması gerekemektedir.

### DYNAMIC & VAR

Var değişkeni içerisinde her çeşit veri alabilmektedir.

var a = 5;

a = 10;

Console.WriteLine( a );

var b = "Ahmet";

b = "Mehmet";

Console.WriteLine(b);

Görüldüğü üzere içerisine atılan her tipten değişkeni alarak işlem yapabilme kabiliyetine sahiptir. Beraberinde ilk farlılık ise ilk atamadan sonra değişken değişse bile farklı bir tipte veri atmaya kalkma sorunsalıdır.

var c = "Ahmet";

c = 3;

Console.WriteLine(b);

Görüldüğü gibi hata verecektir. Fakat dinamik veri tipi buna izin vermektedir.

dynamic variable1 = 1;

Console.WriteLine(variable1);

variable1 = "Furkan"; // en son atadığınız değer ne ise o olur

dynamic variable = new { id = 1, name = "Furkan" };

Console.WriteLine(variable.id + " " + variable.name); // bize yardım etmez değerin adını biz doğru şekilde yazmamız gerekir

dynamic variable2 = new[] {

new { id = 1, name = "Ahmet" },

new { id = 2, name = "Yavuz" },

new { id = 3, name = "Tuğba" }

};

foreach (var item in variable2)

{ Console.WriteLine(item.name); Console.WriteLine("---------"); }

QUERY DYNAMIC

var products = connection.Query("select \* from Products");

// IEnumerable<dynamic> olarak döner

foreach (var item in products)

{

Console.WriteLine("id:" + item.Id);

Console.WriteLine("name:" + item.Name);

Console.WriteLine("price:" + item.Price);

Console.WriteLine("stock:" + item.Stock);

Console.WriteLine("----------------------");

}

// İsimleri doğru yazmanız gerekmektedir. Yardımcı olmaz.

QUERY TİP GÜVENLİ ÇALIŞMAK

internal class Product // internal: ilgili namespace altında erişilebilir demek

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int Stock { get; set; }

}

var products = connection.Query<Product>("SELECT \* FROM Products");

// IEnumerable<Product> oldu

foreach (var product in products)

{

Console.WriteLine("id:" + product.Id);

Console.WriteLine("name:" + product.Name);

Console.WriteLine("price:" + product.Price);

Console.WriteLine("stock:" + product.Stock);

Console.WriteLine("------------------------");

}

// foreach ile gezilebilir bir product listesi dönüyor

QUERYFIRST & QUERYSINGLE & QUERYFIRSTORDEFAULT & QUERYSINGLEORDEFAULT

var product = connection.QueryFirst<Product>("SELECT \* FROM Products where Id=3");

Console.WriteLine(product.Name); // Televizyon

var product2 = connection.QuerySingle<Product>("SELECT \* FROM Products where Id=3"); // sonucun 1 tane olduğundan eminsek querysingle kullanabiliriz

/\* Database olmayan bir veriyi getirmeye çalışalım örnek.

\* QuerySingle ile QueryFirst hata verir

\* QuerySingleOrDefault ile QueryFirstOrDefault hata vermez. Değerler null döner

\*

\* QueryFirst: Siz bir listenin içerisindeki ilk elemanı alabilirsiniz.

\* QuerySingle: Tek bir eleman dönmek zorunda

\* OrDefault ekleniyor ise bir nesne dönmeli ve nesne dönmüyor ise değer null döner

\*/

Console.WriteLine(product2.Name); // Televizyon

JOIN SORGULARI (TABLOLARIN OLUŞTURULMASI)

create table Categories(

Id int identity(1,1),

Name nvarchar(50),

primary key(Id)

)

Categories tablosu oluşturuldu

create table ProductCategory(

Id int identity(1,1),

ProductId int,

CategoryId int,

primary key(Id),

unique(ProductId,CategoryId),

foreign key(ProductId) references Products(Id),

foreign key(CategoryId) references Categories(Id)

)

ProductCategory oluşturuldu.

Product ile Category arasında çoka çok bir ilişki kurmuş olduk.

DAPPER MAP MANTIĞI

var products = connection.Query<Product, ProductCategory, Product>(@"SELECT \* FROM Products

INNER JOIN ProductCategory On Products.Id = ProductCategory.ProductId

WHERE ProductCategory.CategoryId = @categoryId", (product, productCategory) =>

{

return product;

}, new { categoryId = 3 });

foreach (var item in products)

{

Console.WriteLine(item.Name);

}

// 6 of 11 <TFirst,TSecond,TReturn>

// Bir kategorinin birden fazla productcategorisi olabilir

// Bir productın birden fazla productcategorisi olabilir

BAĞLI TABLOYU GETİRMEK

var products = connection.Query<Product>(@"SELECT Product.\* FROM Products

INNER JOIN ProductCategory On Products.Id = ProductCategory.ProductId

where ProductCategory.CategoryId = @categoryId", new { categoryId = 3 });

var productDictionary = new Dictionary<int, Product>();

// key ve value değeri var

var products = connection.Query<Product, ProductCategory, Product>(@"SELECT \* FROM Products INNER JOIN ProductCategory

On Products.Id = ProductCategory.ProductId", (product, productCategory) =>

{ Product otherProduct; if (!productDictionary.TryGetValue(product.Id, out otherProduct))

// product içerisinde id değeri var mı ? {

otherProduct = product; otherProduct.ProductCategories = new List<ProductCategory>(); productDictionary.Add(product.Id, product);

} otherProduct.ProductCategories.Add(productCategory); return otherProduct;

}).Distinct().ToList();

TRANSACTION’I ANLAMA

CREATE TABLE BankAccounts(

Id int identity(1,1),

FullName nvarchar(100),

Money decimal(18,3),

primary key (Id)

)

BankAccounts tablosu oluşturuldu

INSERT INTO BankAccounts(FullName,Money)

VALUES ('Yavuz',5000),

('Ayşe',6000)

BankAccounts tablosuna iki adet kayıt eklendi

connection.Execute("Update BankAccounts Set Money=4500 where Id=1");

Console.WriteLine();

connection.Execute("Update BankAccounts Set Money=6500 where Id=2");

BankAccounts Id değeri 1 olan kişiden para 500 para kesildiği zaman bir kopma olduğunda işlem Id değeri 2 olan üyeye aktarılamayacak. Bu tür işlemlerde Transaction kullanılıyor.

DAPPER TRANSACTION

connection.Open();

var transaction = connection.BeginTransaction();

connection.Execute("Update BankAccounts Set Money=4500 where Id=1",transaction: transaction);

Console.WriteLine();

connection.Execute("Update BankAccounts Set Money=6500 where Id=2", transaction: transaction);

connection.Close();

Artık şimdi Console WriteLine kısmında işleminizde bir hata olduğunda Id=! Olan üyenin parası 4500 olmuyacak ve update işlemi gerçekleşmeyecek yani değişiklikler gerçekleşmemiş olacak. İlla ki başarılı bir şekilde gerçekleşmesini bu şekilde sağlıyoruz. Transaction bu işe yarıyor.

ADD PRODUCT STOREPROCEDURE

CREATE PROCEDURE sp\_addProduct

@name nvarchar(50),

@price decimal(18,3),

@stock int

as

begin

insert into Products values(@name,@price,@stock)

end

UPDATE PRODUCT STOREPROCEDURE

CREATE PROCEDURE sp\_updateProduct

@name nvarchar(50),

@price decimal(18,3),

@stock int,

@id int

as

begin

update Products set Name=@name, Price=@price,Stock=@stock

where Id = @id

end

DELETE PRODUCT STOREPROCEDURE

CREATE PROCEDURE sp\_deleteProduct

@id int

as

begin

delete from Products where Id = @id

end

GET PRODUCT STOREPROCEDURE

CREATE PROCEDURE sp\_getProduct

as

begin

select \* from Products

end

ADD PRODUCT

connection.Execute("sp\_addProduct", new

{

name = "Kitap",

price = 30,

stock = 3000

}, commandType: System.Data.CommandType.StoredProcedure);

UPDATE PRODUCT

connection.Execute("sp\_updateProduct", new

{

name = "Kalem",

price = 25,

stock = 3000,

id = 28

}, commandType: System.Data.CommandType.StoredProcedure);

DELETE PRODUCT

connection.Execute("sp\_deleteProduct", new

{

id = 28

}, commandType: System.Data.CommandType.StoredProcedure);

GET PRODUCT

var products = connection.Query<Product>("sp\_getProduct", commandType: System.Data.CommandType.StoredProcedure);

Console.Write(products.FirstOrDefault().Name); // Ekrana Samsung yazar

DAPPER CONTRIB

Dapper Contrib kullanabilmek için öncelikle Manage Nuget Packages kısmından kurmamız gerekiyor. Dapper Contrib sorgularına erişebilmek için kütüphanelere

using Dapper.Contrib.Extensions;

eklememiz gerekiyor.

Dapper Contrib kullanabilmek için nesnelere ihtiyacımız var.

[Table("Products")]

public class Product

{

[Key]

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int Stock { get; set; }

}

Id değerimizin Key olduğunu belirtmemiz lazım belirtmezsek.

System.Data.DataException:’Get<T> only supports an entity with a [Key] or an [ExplicitKet] propert’ hatasını almış oluruz.

GET

var product = connection.Get<Product>(id: 5);

id değeri 5 olan Product bilgisini getir dedik

GETALL

var products = connection.GetAll<Product>();

Product tablosuna ait tüm değerleri getirmiş olduk

INSERT

connection.Insert<Product>(new Product

{

Name = "Yeni Ürün",

Price = 350,

Stock = 30

});

connection.Insert(new Product

{

Name = "Yeni Ürün 2",

Price = 500,

Stock = 50

});

İki insert işlemi de çalışır

UPDATE

connection.Update(new Product

{

Id = 30,

Name = "Yok",

Price = 7000,

Stock = 700

});

Id değeri 30 olan ürünün bilgilerini değiştirmiş olduk.

DELETE

connection.Delete(new Product

{

Id = 30

});

Id değeri 30 olan ürünü silindi

DELETE ALL

connection.DeleteAll<Product>();

Tüm tabloyu silmiş bulunmaktayız.

CONNECTION’IN MERKEZİ BİR NOKTAYA TAŞINMASI

REPOSITORY DESIGN PATTERN: Kendini tekrar etme

Yaptığın işleri merkezi bir noktaya eklemen gerekir.

Her metotta tekrar tekrar aynı kod içeriğini yazmak sıkıntı ve değişiklik zor olur.

Biz şimdi connection’ı merkezi bir noktaya alacağız.

GENERIC REPOSITORY

Buradaki asıl amaç kodlarımızı olabildiğince bölgelere ayırmak ve merkezi bir noktada oluşturmak böylece tekrar tekrar aynı kodu yazmamız gerekmeyecek.

GetAll(), Add(), Update(), Delete(), GetById()

namespace DapperConsole.Repositories

{

public class GenericRepository<T> where T : class, new()

{

public List<T> GetAll()

{

using (var connection = Db.GetConnection())

{

return connection.GetAll<T>().ToList();

}

}

public void Add(T entity)

{

using (var connection = Db.GetConnection())

{

connection.Insert(entity);

}

}

public void Update(T entity)

{

using (var connection = Db.GetConnection())

{

connection.Update<T>(entity);

}

}

public void Delete(T entity)

{

using (var connection = Db.GetConnection())

{

connection.Delete<T>(entity);

}

}

public T GetById(int id)

{

using (var connection = Db.GetConnection())

{

return connection.Get<T>(id);

}

}

}

}

PRODUCT REPOSITORY

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DapperConsole.Repositories

{

public class ProductRepository : GenericRepository<Product>

{

}

}

KULLANIMI

ProductRepository productRepository = new ProductRepository();

productRepository.Add(new Product()

{

Name = "Dapper",

Price = 5000,

Stock = 100

});

Peki işlem sırasında böyle bir hata aldınız diyelim

**System.Data.SqlClient.SqlException:** 'Invalid object name 'Categorys'.'

Bunun sebebi Dapper maplerken sistem şöyle algılıyor. Direkt olarak Category classına gidiyor sonra gidip sonuna bir s takısı ekliyor. Bizim Category tablomuz Categorys oluyor ve onu database üzerinde arıyor. Bulamadığı için de bize böyle bir hata veriyor. Bu hatayı kaldırmak için Table attribute kullanacağız ancak table attribute hem Data.Annotations da hem de Dapper.Contrib.Extension içinde var bu yüzden karışmaması adına uzun uzun yazmayı tercih ediyorum. Tabi uzun uzun yazmak yerine de using olarak da ekleyebiliriz bu şekilde de kullanımı mevcuttur.

[Dapper.Contrib.Extensions.Table("Categories")]

Tablo adımız ne ise aynısını yazmamız gerekiyor.

Artık bu işlem şöyle oluyor sen Category class’ın karşılığını database üzerinde Categories diye ara. Artık dapper Category gördüğü zaman nereye bakacağını biliyor.

Peki işlem sırasında böyle bir hata aldınız diyelim

**System.Data.SqlClient.SqlException:** 'Cannot insert explicit value for identity column in table 'Categories' when IDENTITY\_INSERT is set to OFF.'

Bunun sebebi Category classının Id propertisini açık değer eklediğimiz için böyle bir hata veriyor. Bu hatayı kaldırmak için Key attribute kullanacağız ancak key attribute hem Data.Annotations da hem de Dapper.Contrib.Extension içinde var bu yüzden karışmaması adına uzun uzun yazmayı tercih ediyorum. Tabi uzun uzun yazmak yerine de using olarak da ekleyebiliriz bu şekilde de kullanımı mevcuttur.

using Dapper.Contrib.Extensions;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DapperConsole.Entities

{

[Dapper.Contrib.Extensions.Table("Categories")]

public class Category

{

[Key]

public int Id { get; set; }

public string Name { get; set; }

}

}

DAPPER CONTRIB – DATA ANNOTATION

Şimdi sıra geldi Dapper.Contrib.Extensions attributelarını öğrenmeye. Table ve Key attribute öğrenmiştik. Şimdi diğer attribute özelliklerini de öğrenelim.

Key: Özelliğin, veritabanı tarafından otomatik olarak oluşturulan bir anahtar olduğunu belirtin (Kimlik Sütunu).

Computed: Özelliğin güncelleme dışında bırakılması gerektiğini belirtir.

ExplicitKey: Özelliğin, veritabanı tarafından otomatik olarak OLUŞTURULMAYAN bir anahtar olduğunu belirtin .

Table: Varlığa eşlenen hedef tablo adını belirtin.

Write: Özelliğin yazılabilir olup olmadığını belirtin.

Computed: Özelliğin güncelleme dışında bırakılması gerektiğini belirtin.

PORTFOLIO APP

Tabloların Oluşturulması

CREATE DATABASE Portfolio

use Portfolio

CREATE TABLE Users(

Id int identity(1,1),

FirstName nvarchar(30),

LastName nvarchar(30),

Address nvarchar(150),

Email nvarchar(50),

UserName nvarchar(25),

ImageUrl nvarchar(50),

Password nvarchar(25),

Phone nvarchar(25),

Description nvarchar(500),

primary key(Id))

CREATE TABLE SocialLMediaIcons(

Id int identity(1,1),

Link nvarchar(200),

Icon nvarchar(20),

UserId int,

primary key(Id))

CREATE TABLE Experiences(

Id int identity(1,1),

Title nvarchar(75),

SubTitle nvarchar(50),

Description nvarchar(500),

StartDate date,

EndDate date,

primary key(Id))

CREATE TABLE Educations(

Id int identity(1,1),

Title nvarchar(75),

SubTitle nvarchar(50),

Grade nvarchar(10),

StartDate date,

EndDate date

primary key(Id)

CREATE TABLE Skills(

Id int identity(1,1),

Description nvarchar(150),

primary key(Id))

CREATE TABLE Interest(

Id int identity(1,1),

Description nvarchar(250),

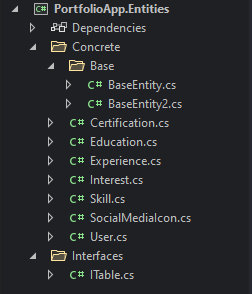
primary key(Id))

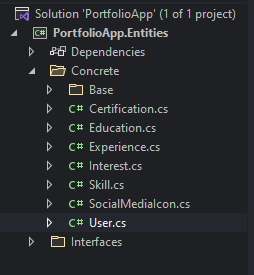
CREATE TABLE Certifications(

Id int identity(1,1),

Description nvarchar(100),

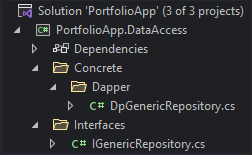
primary key(Id))

Entitylerin Oluşturulması



Interface içerisinde birtane ITable interface var. Base içerisinde BaseEntity ve BaseEntity2 diye classlar var. Base klasörü içerisinde ortak propertyleri olan içerikler konuldu. Örnek olarak Id tüm tablolarda vardı o yüzden ortak bir değer olduğu için eklendi.

DataAccess Oluşturulması



Interface klasörüne IGenericRepository oluşturup işlemlerimizi yapıyoruz.

using PortfolioApp.Entities.Interfaces; // eklememiz gerekiyor.

namespace PortfolioApp.DataAccess.Interfaces

{

public interface IGenericRepository<TEntity> where TEntity : class, ITable, new()

{

List<TEntity> GetAll();

TEntity GetById(int id);

TEntity Insert(TEntity entity);

TEntity Update(TEntity entity);

void Delete(TEntity entity);

}

}

Bu bir class olması gerekiyor. Bir ITable olması gerekiyor. Abstract ve Interface olmaması için new() yazıp newlenebilir diyoruz.

ITable eklediğimizde görmeyebilir bu durumda ilgili katmanda Dependencies --> Add Project Reference diyip Entities ekliyoruz.

Generic Repository Yazılması

using Dapper.Contrib.Extensions;

using PortfolioApp.DataAccess.Interfaces;

using PortfolioApp.Entities.Interfaces;

using System.Collections.Generic;

using System.Data;

using System.Linq;

namespace PortfolioApp.DataAccess.Concrete.Dapper

{

public class DpGenericRepository<TEntity> : IGenericRepository<TEntity> where TEntity : class, ITable, new()

{

private readonly IDbConnection \_connection;

public DpGenericRepository(IDbConnection connection)

{

\_connection = connection;

}

/\*

\* IDbConnection ile ilgili connectionı taşıyabiliyoruz.

\*/

public void Delete(TEntity entity)

{

\_connection.Delete(entity);

}

public List<TEntity> GetAll()

{

return \_connection.GetAll<TEntity>().ToList();

}

public TEntity GetById(int id)

{

return \_connection.Get<TEntity>(id);

}

public TEntity Insert(TEntity entity)

{

\_connection.Insert(entity);

return entity;

}

public TEntity Update(TEntity entity)

{

\_connection.Update(entity);

return entity;

}

}

}

Data Transfer Object

Biz view kısmına yani User Interface e bir şeyler dönerken olduğu gibi buradaki Entityleri dönmememiz gerekiyor. Burada User tablosun ekleme yaparken Id değerine ihtiyacım yok ama update yaparken ihtiyacımız var. Bu tür durumlarda data transfer objectleri kullanıyoruz. Doğrudan hiçbir zaman buradaki Entity gidipte view’e göndermiyoruz.

Örnek Model:

public class User : ITable

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public string ImageUrl { get; set; }

public string Phone { get; set; }

public string UserName { get; set; }

public string Description { get; set; }

public DateTime CreateDate { get; set; }

public bool IsActive { get; set; }

}

DTO Türleri:

GetAll()

Ortak olmayan kısımlar (Password (Password güvenlik için eklenmez))

Add()

Ortak olmayan kısımlar (Id, CreateDate, IsActive (otomatik geldikleri için eklenmez))

Update()

Ortak olmayan kısımlar (Password (Password ayrı bir yerde tutulur ve gelen değer dönüştürülerek tablodaki değer ile kıyaslanır))

GetById()

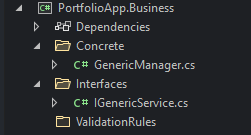
Ortak olmayan kısımlar (Sisteme göre değişir. Tüm değerler lazım ise

public User User {get; set;} ile GetUserDto class’ına eklenir. Arayüz kısmında ise hangi alanlar lazım ise eklenmesi gerekir.)

Değerler birbirleri arasında ortak olsa bile aynı class kullanamayız. Add ve Update kısmı için Validasyon yapılacağı için diğer classlardan ortak değer alamaz. List için Validasyon yapılmaz.

BUSINESS

Generic Service yazılması



Concrete kısmı için bir adet Generic Manager, Interfaces alanı için bir tane IGenericService oluşturduk.

IGenericService içerisi:

namespace PortfolioApp.Business.Interfaces

{

public interface IGenericService<TEntity> where TEntity : class, ITable, new()

{

List<TEntity> GetAll();

TEntity GetById(int id);

TEntity Insert(TEntity entity);

TEntity Update(TEntity entity);

void Delete(TEntity entity);

}

}

GenericManager içerisi:

namespace PortfolioApp.Business.Concrete

{

public class GenericManager<TEntity> : IGenericService<TEntity> where TEntity : class, ITable, new()

{

private readonly IGenericRepository<TEntity> \_genericRepository;

public GenericManager(IGenericRepository<TEntity> genericRepository)

{

\_genericRepository = genericRepository;

}

public void Delete(TEntity entity)

{

\_genericRepository.Delete(entity);

}

public List<TEntity> GetAll()

{

return \_genericRepository.GetAll();

}

public TEntity GetById(int id)

{

return \_genericRepository.GetById(id);

}

public TEntity Insert(TEntity entity)

{

\_genericRepository.Insert(entity);

return entity;

}

public TEntity Update(TEntity entity)

{

\_genericRepository.Update(entity);

return entity;

}

}

}

Validation Rules

Bu yapı için FluentValidation kullanacağız. Neden böyle bir yapı kuruyoruz DataAnnotations ile de yapabiliriz dediğinizi duyar gibiyim evet doğru yapabiliriz ama bu Single Responsibilitye aykırı bir davranış.

Single Responsibility der ki her şeyin bir görevi olmalı. Biz yapımızı dto içerisinde barındıracağımız için. Bu Dto mu yoksa validasyonu mu sağlıyor diyebilir.

Business katmanına FluentValidation.AspNetCore kurulumu yaptıktan sonra FluentValidation klasörü oluşturduk. Yapılacak işlemlerde Dto kullandığımız için DtoValidators adında bir klasör oluşturduk ve classlarımızı yazmaya başladık.

CertificationUpdateDto ile validasyon işlemi yapacağımız için aynı ismi alıp sonuna Validator ekleyerek bir class oluşturduk. Update işlemi yaptığımız için Id değeri boş olmaması gerekiyor. Onun için kurallarımızı aşağıdaki şekilde ayarladık.

using FluentValidation;

using PortfolioApp.DTO.DTOs.CertificationDtos;

namespace PortfolioApp.Business.ValidationRules.FluentValidation.DtoValidators

{

public class CertificationUpdateDtoValidator : AbstractValidator<CertificationUpdateDto>

{

public CertificationUpdateDtoValidator()

{

RuleFor(c => c.Id).NotEmpty().WithMessage("Id alanı boş geçilemez");

RuleFor(c => c.Description).NotEmpty().WithMessage("Sertifika alanı boş geçilemez");

}

}

}

Not: Validasyon kuralları sadece Dto’lar için yazılmaz. Biz bu yüzden sonlarına Dto ekliyoruz yani Dto olduklarını belirtiyoruz. Validasyon kuralları Entityler için de yazılır.

Farkettiniz mi bilmiyorum Id değeri için NotEmpty oluşturduk peki sizce bu doğru mu ? Bu pek doğru bir yöntem değil.

/\* Id değeri hiçbir zaman NotEmpty yani bu validasyona düşmez.

\* Çünkü bu bir integer değer. Integer değerin default hali sıfırdır.

\* Bu işlem için InclusiveBetween ve ExclusiveBetween kullanabiliriz.

\*/

Integer değerin default hali 0 olduğu için Id hiçbir zaman NULL olamıyacağı için validasyona düşmeyecektir.

RuleFor(u => u.Id).InclusiveBetween(1, int.MaxValue).WithMessage("Id boş geçilemez");

Bu yöntemi kullanabiliriz. Id 1 ile int tipin Max değeri arasında olmak zorunda.

IDbConnection

Bu yapıyı Business katmanında gerçekleştireceğiz. Mimariye göre bu yapının Business de olması gerekmektedir. Bu mimariye göre User Interface, Web, WebApi, Mvc veya Mobil katmanın her neyse o katmanın doğrudan DataAccess’i görmemesi gerekiyor. Buradaki amaç hızlı bir şekilde yapıyı değiştirmek.

Business katmanına gidip klasör oluşturuyoruz ve adına IOC(Inversion Of Control) veriyoruz. Klasörün içerisine girip bir klasör daha oluşturup adına Microsoft veriyoruz.

Neden buraya Microsoft yazdık ?

Şu yüzden biz Microsoftun sağlamış olduğu Dependency Injection kütüphanesini kullanacağız. İçerisine bir tane class oluşturup adını ‘ MicrosoftDependencies ’ yapıyoruz.

Biz bir classı genişletmek istiyorsak ilk koşul genişleteceğimizin classın genişletildiğin yerin static olması gerekir. İlgili metodunda static olması gerekir. Static bir class içerisinde zaten static bir metot barındırmak zorundadır. Kullanmaz iseniz zaten hata verir.

this parametresi koyduktan sonra classı genişletmiş olduk

this IServiceCollection services

Biz AddCustomDependencies kullandığımız zaman aslında services kısmını genişletmiş olacağız.

Biz ConnectionString burada yazmayacağız. Bu yüzden bir tane interfaceden faydalanmamız lazım. IConfiguration ile bu işlemi yapacağız

IConfiguration configuration

Configuration.GetConnectionString(“connectionMsSql”) Ben bu IDbConnectiona ilgili sqlConnectionı burada atamış oldum.

namespace PortfolioApp.Business.IOC.Microsoft

{

public static class MicrosoftDependencies

{

public static void AddCustomDependencies(this IServiceCollection services, IConfiguration configuration)

{

services.AddTransient<IDbConnection>(connection => new SqlConnection(configuration.GetConnectionString("connectionMsSql")));

}

}

}

Son hali..

Bağımlılıkları Kaldırma

TypeOf ile bağımlılığımızı belirteceğiz. Sen herhangi bir tipten IGenericRepository gördüğün zaman herhangi bir tipten DdGenericRepository ver.

Herhangi bir tipten IGenericService gördüğünde aynı tipten GenericManagerı ver.

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

services.AddScoped(typeof(IGenericService<>), typeof(GenericManager<>));

IValidator adında bir interfaceden yararlanıyoruz. UserUpdateDto gördüğün zaman UserUpdateDtoValidator kullan.

Sistem hata verirse bu kısımda ya alakasız bir şey veriyorsunuzdur ya da validatorı yanlış yerde yazmış olabilirsiniz.

services.AddTransient<IValidator<UserUpdateDto>, UserUpdateDtoValidator>();

services.AddTransient<IValidator<CertificationAddDto>, CertificationAddDtoValidator>();

services.AddTransient<IValidator<CertificationUpdateDto>, CertificationUpdateDtoValidator>();

services.AddTransient<IValidator<EducationAddDto>, EducationAddDtoValidator>();

services.AddTransient<IValidator<EducationUpdateDto>, EducationUpdateDtoValidator>();

services.AddTransient<IValidator<ExperienceAddDto>, ExperienceAddDtoValidator>();

services.AddTransient<IValidator<ExperienceUpdateDto>, ExperienceUpdateDtoValidator>();

services.AddTransient<IValidator<InterestAddDto>, InterestAddDtoValidator>();

services.AddTransient<IValidator<InterestUpdateDto>, InterestUpdateDtoValidator>();

services.AddTransient<IValidator<SkillAddDto>, SkillAddDtoValidator>();

services.AddTransient<IValidator<SkillUpdateDto>, SkillUpdateDtoValidator>();

services.AddTransient<IValidator<SocialMediaIconAddDto>, SocialMediaIconAddDtoValidator>();

services.AddTransient<IValidator<SocialMediaIconUpdateDto>, SocialMediaIconUpdateDtoValidator>();

Custom Cookie Based Auth

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)

.AddCookie();

Authentication ekledik ve eklerken cookienin default hallerinden authenticationSchema kullan.

Authentication: Bir sisteme giriş yaptığınız zaman authent olmuş olursunuz.

Authorization: Yetkilerinizi belirler.

Cookie Configürasyonu

app.UseAuthentication();

app.UseAuthorization();

Ekliyoruz.

options.Cookie.HttpOnly = true;

options.Cookie.Name = “PortfolioCookie”;

options.SameSite = Microsoft.AspNetCore.Http.SameSiteMode.

options.Cookie.SecurePolicy = CookieSecurePolicy.

Options.ExpireTimeSpan

HttpOnly: Console açıp document.cookie yazdığınız zaman tarayıcının içerisinde ki cookie bilgilerini verir. Bunu kapatmamız lazım o yüzden true diyoruz.

Name: Cookienin tarayıcıda gözükeceği ad.

SameSiteMode türleri:

Lux, None, Strict, Unspecified

Lux: Diğer web sayfalarına cookienin kullanımını açarsınız.

Strict: Bu kullanımı kapatırsınız.

CookieSecurePolicy türleri: (Güvenlikle ilgili protokoller)

Always: Https de çalışır sadece

SameAsRequest: İkisinde de çalışır.

None

ExpireTimeSpan: Cookienin ne kadar süre hayatta kalacağını belirtir.

Kullanımı: TimeSpan.FromDays(20);

Kullanıcının bir kere giriş yaptığı zaman ilgili bilgileri 20 gün boyunca saklanıcak. 20 gün boyunca bu cookieden kullanıcının bilgilerine erişebileceğim.