**Entity Framework Core**

**DERS 1: ORM Nedir?**

- Yazılım uygulamalarında veriler fiziksel olarak veritabanında tutulmaktadır. (SQL Server, Postgre, Oracle..)

- Yazılım ile veritabanı arasında sürekli bir bağlantı üzerinden iletişim sağlanmaktadır.

Ki yazılım dış dünyadan elde ettiği verileri veritabanına işleyebilsin ya da veritabanındaki verileri istediği

zaman elde edebilsin.

- İşte böyle bir durumda yazılım ile veritabanı arasında biraz önce bahsedildiği üzere bir bağlantı kurularak

verisel trafik gerçekleştirilmektedir.

- Yazılım, bu bağlantı üzerinden veritabanına anlayacağı dilden sorgular gönderir ve veritabanı da bu sorgulara

istinaden gerekli verisel işlemleri gerçekleştirir.

- Bu sorgular genellikle SQL dilindedir.

Örnek:

using System.Data.Client;

await using SqlConnection connection = new("Server=localhost, 1433; Database = Northwind; User Id=sa; Password = "1234yapmayın";

await connection.OpenAsync();

SqlCommand command = new("Select \* from Employees", connection);

SqlDataReader dr = await command.ExecuteReaderAsync();

while(await dr.ReadAsync())

{

CW($"{dr["FirstName"]} {dr["LastName"]}");

}

await connection.CloseAsync();

- ORM süreci de burada başlamaktadır.Kodun içerisinde SQL sorguları ve SQL ile ilgili farklı cümleciklerin olması genel anlamda kodun kirlenmesine sebep olmaktadır.

- Bunun dışında geliştiricinin SQL hakkında olan bilgisinin endişe verici olmaması gerekmektedir. Ayrıca veritabanına

gönderilen SQL neticesinde gelen datalar manuel parse edilmek zorunda bu da kodun SQL'e olan bağlılığını arttırdığı gibi

yönetilebilirliği de kısıtlamaktadır.

- Yani uzun lafın kısası kodun içerisinde SQL cümleleri yazmanın ve veritabanından gelen sonuçların manuel

bir şekilde parse edilmesinin büyük projelerde sorunlar çıkacağı aşikardır.

- Bu durum veritabanı bağımlılığı yaratır. SQL Server, Oracle vs. hangisi kullanılıyorsa ona bağlı kalınması

ileride değişimi zorlaştırır.

- Kompleks SQL sorguların manuel bir şekilde oluşturulması gerekir. Geliştirici açısından SQL sorumluluğu beklenir.

- Veritabanı sorgulama neticesinde gelen datalar manuel olarak dönüştürülür. Sorgu sürecinde tablo, kolon vs.

gibi bağımlılıklar olduğu gibi gelen datalarda da aynı bağımlılıklar sürecektir. Gün geldiğinde veritabanını

değiştirmemiz gerekebilir. Bu durumda tüm SQL kodlarını yeni veritabanına göre Refactoring etmemiz gerekmektedir.

- Veritabanında olan değişikliklere uygun bir şekilde kodun da tekrar review edilmesi gereklidir.

Bir kolon adı değiştiğinde ya da bir kolonun herhangi bir kuralı (constraint, validation vs.) değiştiğinde bu

durumdan kodun haberdar olması için bilinçli bir review gerekmektedir.

- Kodu aşırı derecede veritabanı seviyesine indirger. Bu durum tüm gelişmelerin veritabanı ile uyumlu bir

şekilde seyretmesini zorunlu kılar.

- Geliştirilen yazılımın sürecinde tüm veritabanı işlemleri, o anki kullanılan programlama dili ve OOP'nin

hizmetlerinden istifade etmeksizin icra edilir.

**ORM Nedir?**

- Yazılım ve veritabanı arasındaki bağlantı üzerinden sorgular eşliğinde veri transferini OOP nimetlerinden

istifade ederek sağlayabileceği ve böylece kodun da, geliştiricinin de SQL'E bağlılığı olmaksızın hızlı ve kolayca

operasyonları gerçekleştirebileceği bir yaklaşım ortaya konmuştur. Bu yaklaşımın adı ORM'dir.

ORM: Object Relational Mapping - Nesne İlişkisel Eşleme

- Geliştirilen yazılım içerisinde OOP yapısına uygun olmayan, katı ve kompleks veritabanı sorguları yerine veritabanı

objelerinin, bir OOP nesnesi gibi düşünülerek yazılım tarafından kullanabilmesine olanak sağlayan bir yaklaşımdır.

- Bu yaklaşıma göre veritabanı, tablolar ve veriler yazılım tarafında birer nesneye karşılık gelmektedirler.

Böylece tüm veritabanı süreçlerini OOP kavramlayıyla rahatlıkla yönetebilie ve kodu SQL'den arındırabiliriz.

Örnek:

[SQL, MySQL, Oracle, PostgreSQL] -> Context Class {...} -- Tablolara karşılık gelen class'lar----

--veritabanı class'larında referans edilirler.---> Employees Table -> Employees Class {...}

Böylece o veritabanının tablosu olduğu anlaşılmaktadır.

Sorgu süreci:

Context c = new Context();

//Bu veritabanında,

var employeeDatas = c.Employees.ToList();

//Employees tablosunu sorgula ve tüm verileri getir demiş olduk.

- Nihai olarak, yazılım tarafında ORM ile elde edilen tüm bu veriler(employeeDatas) esasında tabloya karşılık

oluşturulmuş olan bu class'ın(Employees) instance'larını(object) olarak elde edilmiş olacaklar.

Örnek:

ORM ile;

NorthwindDbContext context = new();

var employeeDatas = await context.Employees.ToListAsync();

ORM'siz hali;

using System.Data.Client;

await using SqlConnection connection = New("Server=localhost, 1433; Database = Northwind; User Id=sa; Password = "1234yapmayın";

await connection.OpenAsync();

SqlCommand command = new("Select \* from Employees", connection);

SqlDataReader dr = await command.ExecuteReaderAsync();

while(await dr.ReadAsync())

{

CW($"{dr["FirstName"]} {dr["LastName"]}");

}

await connection.CloseAsync();

**ORM Avantajları**

- Veritabanı bağımsızlığı sağlar.

- Kullanılan veritabanına göre uygun sorgu oluşturur.

- OOP nimetlerinden faydalanarak SQL mantığı işlenmesini sağlar.

- Geliştiricinin kullanılan veritabanına dair SQL niteliklerinin olması beklenmez.

- Sorgular otomatik generate edileceğinden dolayı kodu SQL bağımlılığından soyutlar.

**Entity Framework Core Nedir?**

- ORM yaklaşımını benimsemiş bir araçtır.

- Kod içerisinde OOP nimetlerinden istifade ederek SQL sorguları oluşturmamızı sağlamaktadır.

- Açık kaynaktır, esnektir, geliştirilebilir.

- Kod içerisinde ihtiyaca binaen geliştirilmiş olan tekrarlı SQL sorgucuklarından bizi kurtarmaktadır.

- Code First ve Database First yaklaşımları eşliğinde veritabanı ile yazılım arasındaki koordinasyonu sağlamaktadır.

- Kod üzerinden; veritabanı ve tablo, Constraint, Sequence, İlişkili sorgular, View, Stored Procedure, Function, Temporal Table gibi veritabanı nesneleri oluşturmamızı ve kullanmamızı sağlamaktadır.

- Query için LINQ sorgularını desteklemektedir.

**ORM Olarak Neden EF Core Seçilmelidir?**

- EF Core, her ne kadar hızlı ve performanslı bir yapıya sahip olsa da piyasadaki en hızlı ORM aracı değildir.

- Misal olarak; minimal özelliklere sahip olan Dapper, Raw(ham) sorgular kullandığından dolayı kelimenin tam anlamıyla EF Core'dan çok daha hızlıdır.

- Lakin her bir güncellemesinde performansının arttığı gözlemlenen EF Core'un ise birçok özelliği mevcuttur.

- OOP nimetlerinden istifade etmemizi sağlayan EF Core ile class oluşturma, nesne değişikliklerini izleme(Change Tracker), mapping vs gibi türlü işlemi gerçekleştirebiliriz.

- Seviyorum kardeşim.

**EF Core Nasıl Yüklenir?**

- İleride Entity Framework Core ile migration yapılanmalarına vs. değiniyor olacağız. İşte bu yapılanmaları sağlayabilmek ve işlevleri yürütebilmek için belirli araçları

yüklememiz gerekmektedir.

Yüklenecek araçlar: .NET Core command-line interface (CLI) tools

Package Manager Console (PMC) tools

olmak üzere iki farklı kümede değerlendirilebilir.

.NET Core command-line interface (CLI) tools: dotnet-ef ile başlayan CLI komutlarını ilgili PC'de aktif olarak kullanabilmemizi sağlayan tool'dur.

Install -> dotnet tool install --global dotnet-ef

Update -> dotnet tool update --global dotnet-ef

Check -> dotnet ef

Tool'u yükledikten sonra belirli bir projede kullanabilmek için ilgili projede şu paketin de yüklü olması gerekmektedir.

-Microsoft.EntityFramworkCore.Design

Alternatif olarak,

Package Manager Console (PMC) tools

- Visual Studio, Package Manager Console üzerinden talimatlar vermemizi sağlayan bir tool'dur.

Tools -> Nuget Package Manager -> Package Manager Console

Haliyle Package Manager Console üzerinden talimatlar verebilmek için şu paketin ilgili projede yüklü olması gerekmektedir.

-Microsoft.EntityFramworkCore.Tools

**EF Core, Yaklaşımlar**

- EF Core veritabanlarıyla iki farklı yaklaşımı baz alarak çalışmalar sergilemektedir.

Bu yaklaşımlar;

-Database First

-Code First

- Yaklaşım, bir konuyu, olguyu , yapıyı, inşayı, sorunu ve çözümü ele alış bir başka deyişle ona bütünsel olarak bakış biçimidir.

Her yaklaşım bir davranışa özeldir, (öznel)subjektiftir.

- EF Core, veritabanı çalışmaları için, veritabanının önceden olup olmaması durumlarına göre farklı yaklaşımlar sunmaktadır. İşte bu durumlara göre farklı

yaklaşımlar eşliğinde çözüm getirilmesi öznel olmanın bir göstergesidir.

**Yaklaşımların Temel Amacı Nedir?**

- Önceki derslerde ORM yapılanmasını ele alırken, veritabanını OOP nimetlerinden istifade ederek kod tarafında modellememizi sağlayan ve nesneler üzerinden

sorgulama süreçlerini yöneterek SQL'den bağımsız bir şekilde çalışmalar ortaya koyan pattern olduğundan bahsetmiştik.

- İşte bu bilgiye istinaden, bir ORM aracı olan EF Core ile yapılacak olan çalışmalarda da hedef veritabanının kod kısmına aktarılması gerekecektir.

Lakin hedef veritabanı, bazen önceden oluşturulmuş bir veritabanı olabileceği gibi bazen de daha yeni oluşturulacak bir veritabanı da olabilmektedir.

- Yani olan bir veritabanını kod kısmında modellemek gayet gözle görülebilir bir durumken, hiç oluşturulmamış(yeni oluşturulacak olan) bir veritabanını da kod kısmında

modellemek gerekebilmektedir.

- İşte bu yaklaşımlar, veritabanının önceden var olup olmaması durumlarına göre kod kısmında modellenme süreçlerinin hangi davranışla gerçekleştirileceğini

beirleyecek olan tekniği bizlere sunmaktadırlar.

- Misal olarak uzun zamandır devam eden ve veritabanı mevcut bir projeye katıldığınızı varsayalım.

Bu durumda projedeki veritabanının var olmasından dolayı büyük ihtimalle Database First yaklaşımını tercih etmemiz gerekecektir.

- Lakin veritabanı daha inşa edilmemiş bir proje ise böyle durumlarda Database First veya Code First yaklaşımlarından birini tercih edecebiliriz.

**Database First Yaklaşımı**

- EF Core ile çalışma yapılacak olan veritabanı önceden oluşturulmuş ve içerisine tablolar yerleştirilerek belirli çalışmalar yapılmış ise bu veritabanını kod

kısmında modellemek için en doğru yaklaşım Database First yaklaşımıdır.

- Database First, var olan veritabanını tersine mühendislik ile analiz edip otomatik olarak kod kısmında modelleyen bir yaklaşımdır. Yani şöyle ki, hedef veritabanının belirli talimatlar aracılığıyla otomatik olarak kod kısmında OOP nimetleri eşliğinde modellenmesidir.

Önceden oluşturulmuş veritabanı --SQL--> Tablolar (Personeller, Müşteriler, Satışlar, Ürünler)

Bu veritabanını kod tarafında modelleyebilmek için ileride Scaffold adını vereceğimiz bir talimatı kullanacağız.

--Scaffold Talimatı--> Kod Ortamı

//veritabanına karşılık gelen sınıf

class DbContext

{

//aşağıdaki 4 sınıfın veritabanı sınıfında nesnel olarak tabloları olduğunu ifade eden referansları;

public DbSet <Personeller> Personeller {get; set;}

public DbSet <Satışlar> Satışlar {get; set;}

public DbSet <Müşteriler> Müşteriler {get; set;}

public DbSet <Ürünler> Ürünler {get; set;}

//belirlemek gerekir.

}

//tablolara karşılık gelen sınıflar

//her bir kolon bir property'e karşılık gelir.

class Personeller {Adi, Soyadi..}

class Satışlar {Personel ID,}

class Müşteriler {Adi, Soyadi}

class Ürünler {ÜrünAdi, Adet}

**Database First Yaklaşımının Avantajları ve Dezavantajları**

Avantajları;

- Hazır veritabanlarını hızlı bir şekilde modelleyebilmemizi sağlar.

- Veritabanında süreçte olan değişiklikleri de hızlıca koda aktarmamızı sağlar.

- SQL Server, Oracle, PostgreSQL vs. gibi EF Core tarafından desteklenen tüm veritabanlarında kullanılabilir.

- Veritabanından bağımsız olarak tüm modellemeyi OOP nimetleri karşılığında sağlamaktadır.

Dezavantajları;

- Kod veritabanına göre şekilleneceği/modelleneceği için yönetim veritabanı tarafından sağlanır. Haliyle veritabanı bilgisi gerektirir.

- Değişiklikler veritabanı kısmında olacağı için geliştirici tarafından sürekli bir kontrol/güncelleme davranışı sergilenmelidir.

Hangi Durumlarda Tercih Edilmelidir?

- Önceden oluşturulmuş, hali hazırda veritabanı var olan uygulamalarda,

- Uzun süreli devam eden uygulamalarda(özellikle devlet gibi köklü kurumsal projelerin veritabanlarını modellerken),

- Veritabanı yönetimine, geliştirme süreçlerine ve tasarımına dair herhangi bir kararın geliştiriciler tarafından verilmediği durumlarda tercih edilmelidir.

**Code First Yaklaşımı**

- EF Core çalışma yapılacak olan veritabanı önceden oluşturulmamış ise bu veritabanını kod kısmında modelleyerek ardından bu modele uygun veritabanını sunucuda oluşturtan(migration) yaklaşımıdır.

- Bu yaklaşımda veritabanı önce kodla tasarlanır, sonra veritabanı sunucusuna gönderilecek veritabanı oluşturulur.

- Database First yaklaşımının tam tersi davranışı gerektirir.

class DbContext

{

public DbSet <Personeller> Personeller {get; set;}

public DbSet <Satışlar> Satışlar {get; set;}

public DbSet <Müşteriler> Müşteriler {get; set;}

public DbSet <Ürünler> Ürünler {get; set;}

}

class Personeller {Adi, Soyadi..}

class Satışlar { Personel ID, }

class Müşteriler {Adi, Soyadi}

class Ürünler {ÜrünAdi, Adet}

----migrate--->

Kod kısmında tasarlanan veritabanı modeli, veritabanı sunucusuna gönderildiğinde(ki bu işleme migrate denir)

nihai olarak tasarlanan modele karşılık veritabanı oluşturulmuş olacaktır.

--> Veritabanı --SQL--> Tablolar (Personeller, Müşteriler, Satışlar, Ürünler)

- İlgili sınıflara karşılık tablolar oluşturulacak, sınıfların içerisindeki property'ler kolon olarak eklenecektir.

Avantajları;

- Kod üzerinden veritabanını modellememizi sağlar.

- Veritabanına dokunmaksızın kod üzerinden gerekli düzenlemeleri ve güncellemeleri hızlıca yapabilmemizi sağlar.

- SQL Server, Oracle, PostgreSQL vs. gibi EF Core tarafından desteklenen tüm veritabanlarında kullanılabilir.

- Veritabanından bağımsız olarak tüm modellemeyi OOP nimetleri karşılığında sağlamaktadır.

- Koddaki ihtiyaca dönük veritabanı şekilleneceği/modelleneceği ve nihai olarak bu modele göre inşa edileceği için yönetim geliştirici tarafından sağlanmaktadır.

Haliyle herhangi bir veritabanı bilgisine gerek duymamaktadır.

- Değişiklikler kod kısmından yapılacağı için geliştirici tarafından sürekli bir kontrol/güncelleme davranışı sergilenmeyecektir.

- Veritabanı modeli kod üzerinde yapıldığı için istenilen sunucuda anında ilgili modeldeki veritabanı elde edilebilir.

Dezavantajları;

- Üretilecek veritabanının tasarımı ve stratejisi geliştirici sorumluluğundadır.

Hangi Durumlarda Tercih Edilmelidir?

- Veritabanı bilgisine ihtiyaç duyulmayan, veritabanı tasarımının kod üzerinden yapılacaki geliştirici tarafndan sorumluluğun üstlenilebileceği,

- Veritabanı yönetiminin kod üzerinden sağlanacağı durumlarda tercih edilebilir.

**Not:** Bu iki yaklaşımın kullanım alanlarının radikal olarak ayrıldığı tek nokta ilgili veritabanının önceden oluşturmuş olup olmamasıdır. Eğer önceden oluşturulmuş ise

Database First yaklaşımı elzemdir.

Yok eğer önceden oluşturulmamış ise Code First yaklaşımını tercih edebileceğiniz gibi önce sunucuda veritabanını manuel tasarlayıp sonrasında da Database First

yaklaşımıyla kod kısmında inşa edebilirsiniz.

İşte burada bir opsiyonellik söz konusudur.

**DERS 2: EF Core Aktörleri**

- Bir ORM aracının veritabanını OOP nimetleriyle temsil edebilmesi için veritabanının, o veritabanının içerisindeki tabloların ve o tabloların içerisindeki kolon ve nesnelerin programatik olarak bir şekilde modellenmesi gerekmektedir.

Bu modelleme class'lar üzerinden gerçekleşecektir.

Veritabanı Nesnesi - DbContext

- EF Core'da veritabanını temsil edecek olan sınıf DbContext olarak nitelendirilmektedir.

Database Northwind -> public class NorthwindDbContext : DbContext { }

Bir class'ın adında DbContext geçmesi yeterli değildir. Bir class'ın veritabanına karşılık gelen DbContext olabilmesi için Microsoft.EntityFrameworkCore namespace'indeki

DbContext sınıfından türemesi gerekmektedir.

DbContext Nesnesinin Sorumlu Olduğu Faaliyetler Nelerdir?

- Konfigürasyon: Veritabanı bağlantısı, model yapılanmaları ve veritabanı nesnesi ile tablo nesneleri arasındaki ilişkileri sağlar.

- Sorgulama: Kod tarafından gerçekleştirilen adımlarını SQL sorgusuna dönüştürür ve veritabanına gönderir.

- Change Tracking: Sorgulama neticesinde elde edilen veriler üzerindeki değişiklikleri takip eder.

- Veri Kalıcılığı: Verilerin kayıt edilmesi, güncellenmesi ve islinmesi operasyonlarını gerçekleştirir.

- Caching

Tablo Nesnesi - EntityFrameworkCore

- EF Core'da tabloları temsil edecek sınıflar Entity olarak nitelendirilmektedir.

- Yeryüzündeki herhangi bir olguyu/nesneyi/objeyi modelleyen sınıfa Entity(varlık) denmektedir. EF Core açısından baktığımızda entity, bir veritabanı tablosunu modelleyen sınıftır.

Northwind(Database)

(Table)Employees [EmployeeID,FirstName,LastName]

(Table)Orders [OrderID,CustomerID,OrderDate]

(Table)Categories [CategoryID,Name,Description]

(Table)Customers [CustomerID,Name,ContactTitle]

(Table)Suppliers [SupplierID,Name,Contact]

(Table)Products [ProductID,Name,SupplierID]

--> Kod ortamı

public class NorthwindDbContext:DbContext

{

//Veritabanında şu entity(DbSet<T>) modeline karşılık bir tablo bulunmaktadır.

//Bu tabloyu şu property ile temsil et demiş oluyoruz.

public DbSet<Employee> Employees {get; set;}

//Employee entity'sine karşılık gelen tabloyu temsil eden property'nin adı Employees

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

public DbSet<Category> Categories {get; set;}

public DbSet<Customer> Customers {get; set;}

public DbSet<Supplier> Suppliers {get; set;}

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

//Ayrıca DbSet property isimlerinin çoğul olduğuna dikkat ediniz.

//Entity sınıfları tekil isimken, bu entity sınıfı türüne karşılık gelen tabloyu temsil eden property çoğul isimli olmalıdır.

}

public class Supplier { }

public class Product { }

public class Employee { }

public class Order { }

public class Category { }

public class Customer { }

- Veritabanında tablo adı çoğul olur, lakin o tabloyu modelleyen entity sınıfının adı tekil olur.

- Tüm entity sınıfları DbContext sınıfı içerisine DbSet olarak eklenmelidir. Böylece veritabanı sınıfı ile entity sınıfları arasında bir ilişki kurulacak ve EF Core temsil ettiğini bu ilişki üzerinden anlayacaktır.

**Tablo Kolonları**

(Table)Customers ->

public class Customer

{

public int CustomerID {get; set;}

public int Name {get; set;}

public int ContactTitle {get; set;}

}

Veriler;

- Veritabanındaki veriler ise entity'lerin instance'larına karşılık gelmektedir.

Customers

CustomerID Name ContactTitle

1 Hilmi Müdür

2 Cafer Çaycı

3 Faruk Pazarlayıcı

new Customer();

CustomerID: 1

Name: Hilmi

ContactTitle: Müdür

new Customer();

CustomerID: 2

Name: Cafer

ContactTitle: Çaycı

new Customer();

CustomerID: 3

Name: Faruk

ContactTitle: Pazarlayıcı

**Tersine Mühendislik (Reverse Engineering)**

- Bir sunucudaki veritabanının iskelesini kod kısmında oluşturma sürecidir.

- Bu süreci Package Manager Console (PMC) ya da Dotnet CLI aracılığıyla iki farklı şekilde yürütebiliriz.

PMC veya Dotnet CLI ile Tersine Mühendislik

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.[Provider]

- Connection String'i bulmak için;

https://www.connectionstrings.com/sql-server/ (MsSQL için)

Standart Security: Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;

- PMC ile veritabanını modelleyebilmek için aşağıdaki kütüphanelerin projeye yüklenmesi gerekmektedir;

Microsoft.EntityFrameworkCore.Tools

Database Provider (Örneğin; Microsoft.EntityFrameworkCore.SQLServer)

- Connection String'de belirttiğimiz ile aynı olmalıdır.

Database Provider bulmak için;

https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

Projemize sağ tıklayıp Manage NuGet Packages diyerek aşağıdaki 2 dosyayı indirelim;

- Microsoft.EntityFrameworkCore.Tools ya da .Design (Dotnet CLI)

- Microsoft.EntityFrameworkCore.SqlServer

Sonra PMC'a gelip;

Scaffold-DbContext 'Server=PC\SQLEXPRESS;Database=NORTHWND;User ID=sa;Password=\*\*\*;TrustServerCertificate=True' Microsoft.EntityFrameworkCore.SqlServer

Ya da Dotnet CLI ile; (powershell üzerinden de yapılabilir)

dotnet ef dbcontext scaffold 'Server=PC\SQLEXPRESS;Database=NORTHWND;User ID=sa;Password=\*\*\*;TrustServerCertificate=True' Microsoft.EntityFrameworkCore.SqlServer

**Tabloları Belirtme**

- Varsayılan olarak veritabanındaki tüm tablolar modellenir. Sadece istenilen tabloların modellenebilmesi için aşağıdaki gibi talimatların verilmesi yeterlidir.

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.SqlServer -Tables Table1,Table2

Ya da Dotnet CLI ile;

dotnet ef dbcontext scaffold 'Connection String' Microsoft.EntityFrameworkCore.SqlServer --table Table1 --table Table2

**DbContext Adını Belirtme**

- Scaffold ile modellenen veritabanı için oluşturulacak context nesnesi adını veritabanından alacaktır. Eğer ki context nesnesinin adını değiştirmek

istiyorsanız aşağıdaki gibi çalışabilirsiniz.

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.SqlServer -Context ContextName

dotnet ef dbcontext scaffold 'Connection String' Microsoft.EntityFrameworkCore.SqlServer --context ContextName

**Path ve Namespace Belirtme**

- Entity'ler ve DbContext sınıfı, default olarak direkt projenin kök dizinine modellenir ve projenin varsayılan namespace'ini kullanırlar. Eğer ki bunlara müdahele etmek istiyorsanız aşağıdaki gibi talimat verebilirsiniz.

Path:

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.SqlServer -ContextDir Data -OutputDir Models

dotnet ef dbcontext scaffold 'Connection String' Microsoft.EntityFrameworkCore.SqlServer --context-dir Data --output-dir Models

Namespace:

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.SqlServer

-Namespace YourNamespace -ContextNamespace YourNamespace

dotnet ef dbcontext scaffold 'Connection String' Microsoft.EntityFrameworkCore.SqlServer --context-dir Data --output-dir Models

--namespace YourNamespace

--context-namespace YourContextNamespace

**Model Güncelleme**

- Veritabanında olan değişiklikleri kod kısmına yansıtabilmek için Scaffold talimatını tekrar vermeniz gerekmektedir lakin verilen talimat neticesinde

ilgili sınıfların zaten var olduğuna dair hata mesajı verilme olasılığı yüksektir.

Böyle bir durumda veritabanı modeline değişiklikleri manuel olarak yansıtabileceğimiz gibi(not recommended), dosyalar var olsa dahi yeniden en güncel haliyle

modellenmesini sağlayabiliriz. Bunun için aşağıdaki gibi Force talimatının verilmesi yeterli olacaktır.

Scaffold-DbContext 'Connection String' Microsoft.EntityFrameworkCore.SqlServer -ContextDir Data -Force

dotnet ef dbcontext scaffold 'Connection String' Microsoft.EntityFrameworkCore.SqlServer --context-dir Data --force

**Modellerin Özelleştirilmesi**

- Database First yaklaşımında veritabanı nesneleri otomatik olarak modellenmekte ve generate edilmektedir. Bazen bu otomatize olan süreçte manuelde olsa entity'lerde yahut context nesnesinde özelleştirmeler yapmak isteyebiliriz.

- Veritabanında yapılan değişiklikler neticesinde Force komutu eşliğinde tüm değişiklikler kod kısmına sıfırdan yansıtılabilir ve bu da yapılan değişikliklerin ezilme riskiin söz konusu olduğu anlamına gelir.

- Bu tarz özelleştirme durumlarında bizzat model sınıflarını kullanmaktansa bunların partial class'ları üzerinde çalışmak en doğrusudur.

**Code First Yaklaşımı**

Migration & Migrate Kavramları

Project

public class DbContext

{

public ... {get; set; }

public ... {get; set; }

public ... {get; set; }

}

public class EntityA

{

public ... {get; set; }

public ... {get; set; }

public ... {get; set; }

}

public class EntityB

{

public ... {get; set; }

public ... {get; set; }

public ... {get; set; }

}

----Migration--> Kod kısmında modellediğimiz veritabanını, veritabanı sunucusunun anlayacağı hale getiren bir C# class'ıdır.

public partial class migration

{

public override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable..

}

public override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable..

}

}

---Migrate---> Veritabanı sunucusunda o modele karşılık bir veritabanının oluşturulma süreci.

Database

(Table)

+SupplierID

+Name

+Contact..

(Table)

+SupplierID

+Name

+Contact..

(Table)

+SupplierID

+Name

+Contact..

**Migration Oluşturmak İçin Temel Gereksinimler**

- Migration oluşturmak için temelde EF Core aktörleri olan DbContext ve Entity class'larını oluşturmak gerekir. Bunları oluşturduktan sonra migration Package Manager Console ve Dotnet CLI olmak üzere iki türlü talimatla verilebilir.

- Her iki yöntemi de kullanabilmek için Microsoft.EntityFrameworkCore.Tools kütüphanesini uygulamaya yüklemeniz gerekmektedir.

Örnek:

using Microsoft.EntityFrameworkCore;

Console.WriteLine("Hello");

public class ECommerceDbContext : DbContext //ECommerceDbContext diye bir veritabanı modellemiş olduk.

{

public DbSet<Product> Products { get; set; } //ve bu veritabanı içerisinde Products isminde bir tablo olacağını,

//bu tablonun modelinin de Product türünde olacağını belirttik.

public DbSet<Customer> Customers { get; set; } //Artık EF Core açısından bir Entity modelidir.

//bu veritabanı modellemesi hangi veritabanı sunucusuna uygun bir şekilde modellendi?

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) //context nesnemizle ilgili temel ayarlamalarımızı yapmamızı sağlayan bir fonksiyondur.

{

//context'in hangi veritabanı sunucusuna migrate edileceği burada belirtilir.

optionsBuilder.UseSqlServer("Server=PC\\SQLEXPRESS;Database=ECommerceDb;User Id=sa;Password=1");

//NuGet üzerinden kütüphaneyi yüklemeyi unutmayalım.

}

}

//Entity

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public int Quantity { get; set; }

public float Price { get; set; }

}

//Entity

public class Customer

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

}

**Migration Oluşturma**

PMC

add-migration [Migration Name]

Dotnet CLI

dotnet ef migrations add [Migration Name]

Mini Not:

oluşturulan mig\_1.cs'e bakacak olursak;

Up : O ana kadar yapılmış çalışmalarda veritabanı sunucusuna gönderilecek olanları belirler,

Down: Migration geri alınırsa yapılmış olanları geri almaya yarar.

**Migration Path'i Belirleme**

PMC

add-migration [Migration Name] -OutputDir [Path]

Dotnet CLI

dotnet ef migrations add [Migration Name] --output-dir [Path]

**Migration Silme**

PMC

remove-migration

Dotnet CLI

dotnet ef migrations remove

**Migration'ları Listeleme**

PMC

get-migration

Dotnet CLI

dotnet ef migrations list

**Migration'ları Migrate Etme (Up Fonsiyonu)**

PMC

update-database

Dotnet CLI

dotnet ef database update

**Migration'ları Geri Alma (Down Fonksiyonu)**

PMC

update-database [Migration Name]

Dotnet CLI

dotnet ef database update [Migration Name]

- Hangi migration'a geri dönülmek istiyorsa o yazılır.

**Kod Üzerinden Migrate Operasyonu**

- Migration'ları tool aracılığıyla migrate edebildiğimiz gibi kod üzerinden de uygulamanın ayakta olduğu süreçte(runtime'da) veritabanını migrate edebiliyoruz.

AppDbContext context = new();

await context.Database.MigrateAsync();

Not:

- Veritabanı üzerinde yapılan tüm değişiklikleri migration eşliğinde gönderiniz. Böylece her bir değişiklikleri migration'lar ile kayıt altına almış

olursunuz(bu da size veritabanı gelişim sürecini verir.) ve ihtiyaca binaen istediğiniz noktaya geri dönüş sağlayabilirsiniz.

- Migration'lara mümkün mertebe dokunmamak lazım. Lakin ileride ihtiyaç doğrultusunda ham SQL cümlecikleri ekleyeceğimiz ve hatta Stored Procedure gibi

yapıları oluşturacağımız noktalar olacaktır.

**DERS 3: OnConfiguring ile Konfigürasyon Ayarlarını Gerçekleştirmek**

- EF CORE tool'unu yapılandırmak için kullandığımız bir metottur.

- Context nesnesinde override edilerek kullanılmaktadır.

using Microsoft.EntityFrameworkCore;

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{

public DbSet<Urun> Urunler { get; set; } //Bu veritabanı içerisinde de Urun entity'si modelinde bir tablo var adı da Urunler.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

//Provider

//ConnectionString

//Lazy Loading vb.

//context nesnesinin hangi veritabanına uygun bir biçimde bir çalışma sergileyeceği belli oldu.

optionsBuilder.UseSqlServer("Server=PC\\SQLEXPRESS;Database=ETicaretDB;User ID=sa;Password=1");

}

}

public class Urun //Entity isimleri tekil.

{

}

**Basit Düzeyde Entity Tanımlama Kuralları**

- Yukarıdaki koda uygun bir migration oluşturmaya çalıştığımızda "The entity type 'Urun' requires a primary key to be defined..." hatası alırız.

- EF Core, her tablonun default olarak bir primary key kolonu olması gerektiğini kabul eder. Haliyle bu kolonu temsil eden bir property tanımlamadığımız taktirde hata verecektir.

using Microsoft.EntityFrameworkCore;

Console.WriteLine("Hello");

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{

public DbSet<Urun> Urunler { get; set; } //Bu veritabanı içerisinde de Urun entity'si modelinde bir tablo var adı da Urunler.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

//Provider

//ConnectionString

//Lazy Loading vb.

//context nesnesinin hangi veritabanına uygun bir biçimde bir çalışma sergileyeceği belli oldu.

optionsBuilder.UseSqlServer("Server=PC\\SQLEXPRESS;Database=ETicaretDB;User ID=sa;Password=1");

}

}

public class Urun //Entity isimleri tekil.

{

//public int Id { get; set; }

//public int ID { get; set; }

//public int UrunId { get; set; }

public int UrunID { get; set; }

//Yukarıdakilerden birini tanımladığımız zaman EF Core tanıyıp primary key olarak belirleyecektir.

//İleride bu şekilde tanımlanan property'ler dışındaki bir property'i de primary key atamayı göreceğiz.

}

- Bu şekilde add-migration mig\_1 yazıp enter'a bastığımızda hata ortadan kalkacaktır.

**Veri Nasıl Eklenir?**

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new(); //veri ekleyebilmek için bir context nesnesine ihtiyacımız var.

//Verilere Entity'lerin instance'ları karşılık gelir.

Urun urun = new() //Artı olarak, Urun adlı entity sınıfından bir nesne yani veri oluşturmamız da lazım.

{

UrunAdi = "A Ürünü",

Fiyat = 1000

};

//context.AddAsync Fonksiyonu kullanabiliriz,

//await context.AddAsync(urun); //bir urun ekleme talimatını vermiş olduk.

//sadece Add'i de kullanabiliriz.

//context.DbSet.AddAsync Fonksiyonu da kullanabiliriz. Tek farkı tip güvenliği sağlamasıdır.

await context.Urunler.AddAsync(urun);

//context nesnesi üzerinden ilgili nesneyi AddAsync ile ekliyoruz.

//burada bir insert işlemi yapmış olduk.

//SaveChanges ya SaveChangesAsync; insert, update ve delete sorgularını oluşturup bir transaction eşliğinde veritabanına //gönderip execute eden fonksiyondur.

//Eğer ki oluşturulan sorgulardan herhangi biri başarısız olursa tüm işlemleri geri alır(rollback).

await context.SaveChangesAsync();

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{

public DbSet<Urun> Urunler { get; set; } //Bu veritabanı içerisinde de Urun entity'si modelinde bir tablo var adı da Urunler.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun //Entity isimleri tekil.

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

}

- Kodu her çalıştırdığımızda, 2 kere çalıştırırsak örneğin;

Id UrunAdi Fiyat

1 A Ürünü 1000

2 A Ürünü 1000

olarak SQL'e eklenecektir.

**EF Core Açısından Bir Veriye Göre Hangi Davranışın Gerektiği Nasıl Anlaşılıyor?**

using Microsoft.EntityFrameworkCore;

ETicaretContext context= new ETicaretContext();

Urun urun = new Urun()

{

UrunAdi = "B Ürünü",

Fiyat = 2000

};

Console.WriteLine(context.Entry(urun).State); //Detached yazacaktır.

//await context.AddAsync(urun);

//await context.SaveChangesAsync();

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{...}

public class Urun //Entity isimleri tekil.

{...}

- Fakat AddAsync'deki yorumu kaldırırsak;

using Microsoft.EntityFrameworkCore;

ETicaretContext context= new ETicaretContext();

Urun urun = new Urun()

{

UrunAdi = "B Ürünü",

Fiyat = 2000

};

Console.WriteLine(context.Entry(urun).State); //Detached yazacaktır. Çünkü bir işlem yapılmadı.

await context.AddAsync(urun);

Console.WriteLine(context.Entry(urun).State); //Added yazacaktır. Çünkü artık veritabanına eklendi.

//await context.SaveChangesAsync();

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{...}

public class Urun //Entity isimleri tekil.

{...}

Çıktı:

Detached

Added

olacaktır.

- SaveChanges'ı da yorumdan çıkarırsak;

using Microsoft.EntityFrameworkCore;

ETicaretContext context= new ETicaretContext();

Urun urun = new Urun()

{

UrunAdi = "B Ürünü",

Fiyat = 2000

};

Console.WriteLine(context.Entry(urun).State); //Detached yazacaktır. Çünkü bir işlem yapılmadı.

await context.AddAsync(urun);

Console.WriteLine(context.Entry(urun).State);

await context.SaveChangesAsync();

Console.WriteLine(context.Entry(urun).State);

public class ETicaretContext : DbContext //ETicaretContext diye veritabanına karşılık gelen bir sınıfım var.

{...}

public class Urun //Entity isimleri tekil.

{...}

Çıktı:

Detached

Added

Unchanged

- SaveChanges'ı çağırıp veritabanına gittikten sonra üzerinde herhangi bir değişiklik yapılmadığı için Unchanged yazdı.

- Silme işlemi yapsaydık Deleted, güncelleme yapsaydık Modified yazacaktı.

**Birden Fazla Veri Eklerken Nelere Dikkat Edilmelidir?**

- SaveChanges'ı verimli kullanmalıyız. SaveChanges fonksiyonu her tetiklendiğinde bir Transaction oluşturacağından dolayı EF Core ile yaılan her bir işleme özel olarak kullanmaktan kaçınmalıyız. Çünkü her işleme özel transaction veritabanı açısından ekstra maliyet demektir. O yüzden mümkün mertebe tüm işlemlerimizi tek bir transaction eşliğinde veritabanına gönderebilmek için SaveChanges'ı aşağıdaki gibi tek seferde kullanmak hem maliyet hem de yönetilebilirlik açısından faydalı olacaktır.

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new ETicaretContext();

Urun urun1 = new Urun()

{

UrunAdi = "C Ürünü",

Fiyat = 2000

};

Urun urun2 = new Urun()

{

UrunAdi = "D Ürünü",

Fiyat = 2000

};

Urun urun3 = new Urun()

{

UrunAdi = "E Ürünü",

Fiyat = 2000

};

await context.AddAsync(urun1);

await context.AddAsync(urun2);

await context.AddAsync(urun3);

await context.SaveChangesAsync(); //böylece tek bir transaction içerisinde yukarıdaki işlemler gerçekleştirilip maliyetten düşülecektir.

public class ETicaretContext : DbContext

{...}

public class Urun

{...}

- AddRange fonksiyonunu da kullanabiliriz.

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new ETicaretContext();

Urun urun1 = new Urun()

{

UrunAdi = "C Ürünü",

Fiyat = 2000

};

Urun urun2 = new Urun()

{

UrunAdi = "D Ürünü",

Fiyat = 2000

};

Urun urun3 = new Urun()

{

UrunAdi = "E Ürünü",

Fiyat = 2000

};

await context.Urunler.AddRangeAsync(urun1, urun2, urun3);

await context.SaveChangesAsync();

public class ETicaretContext : DbContext

{...}

public class Urun

{...}

**Eklenen Verinin Generate Edilen Id'sini Elde Etme**

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new ETicaretContext();

Urun urun = new Urun()

{

UrunAdi = "O Ürünü",

Fiyat = 2000

};

await context.AddAsync(urun);

await context.SaveChangesAsync();

Console.WriteLine(urun.Id);

public class ETicaretContext : DbContext

{...}

public class Urun

{...}

- Ekrana o anki eklenen ürünün Id'sini getirecektir.

**DERS 4: Veri Nasıl Güncellenir?**

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new();

Urun urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 3); //vermiş olduğumuz şarta uyan ilk veriyi getirir.

//önce bu şekilde veriyi context üzerinden bir sorgulama ile elde ettik.

//şimdi nesne üzerinden güncelleyebiliriz.

urun.UrunAdi = "H Ürünü";

urun.Fiyat = 999;

//bu değişikliği veritabanına da bildirmemiz gerek.

await context.SaveChangesAsync();

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

}

**ChangeTracker Nedir? Kısaca.**

- ChangeTracker, context üzerinden gelen verilerin takibinden sorumlu bir mekanizmadır.

- Bu takip mekanizması sayesinde context üzerinden gelen verilerle ilgili işlemler neticesinde update yahut delete sorgularının oluşturulacağı anlaşılır.

**Takip Edilmeyen Nesneler Nasıl Güncellenir?**

- EF Core'da Entity'ler üzerinden aşağıdaki şekilde oluşmuş olduğumuz nesneler direkt veritabanından elde edilmediği için, veritabanındaki nesne ile eşleştirilmez.

çünkü bu nesne context üzerinden bir sorgulama neticesinde gelmedi. Bu yüzden ChangeTracker devreye girmedi.

- Bu takip edilmeyen nesneyi EF Core üzerinden güncellemek istiyorsak Update fonksiyonunu kullanabiliriz.

ETicaretContext context = new ETicaretContext();

Urun urun = new()

{

Id = 3,

UrunAdi = "Yeni Ürün",

Fiyat = 123

}; //aşağıda Update ile güncelleyeceğiz.

**Update Fonksiyonu**

- ChangeTracker mekanizması tarafından takip edilmeyen nesnelerin güncellenebilmesi için Update fonksiyonu kullanılır.

- Update fonksiyonunu kullanabilmek için kesinlikle ilgili nesnede Id değeri verilmelidir!

- Bu değer güncellenecek(update sorgusu oluşturulacak) verinin hangisi olduğunu ifade edecektir.

//takip olmadığından dolayı SaveChanges'ı çağıramayız.

//Update ile EF Core'u bildirip,

context.Urunler.Update(urun);

//öyle SaveChanges'u çağırıyoruz.

await context.SaveChangesAsync();

#endregion

#endregion

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

}

**Entity State Nedir?**

//Bir entity instance'nın durumunu ifade eden bir referanstır.

ETicaretContext context = new();

Urun u = new();

Console.WriteLine(context.Entry(u).State); //Detached yazacaktır.

**EF Core Açısından Bir Verinin Güncellenmesi Gerektiği Nasıl Anlaşılıyor?**

ETicaretContext context = new ETicaretContext();

Urun urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 3);

Console.WriteLine(context.Entry(urun).State); //Unchanged yazacaktır.

urun.UrunAdi = "Domestos";

Console.WriteLine(context.Entry(urun).State); //Modified yazacaktır.

await context.SaveChangesAsync();

Console.WriteLine(context.Entry(urun).State); //Tekrardan Unchanged olacktır.

**Birden Fazla Veri Güncellenirken Nelere Dikkat Edilmelidir?**

ETicaretContext context = new ETicaretContext();

var urunler = await context.Urunler.ToListAsync();

//ilgili DbSet'e karşılık gelen tabloya bir select sorgusu atıp tüm elemanlarını getir.

foreach (var urun in urunler)

{

urun.UrunAdi += "\*"; //her bir ürünün adının yanına \* ekleyelim.

//await context.SaveChangesAsync(); her iterasyonda çağırır ve çok maliyetli olur. Her bir ürüne karşılık bir transaction oluşturur.

}

await context.SaveChangesAsync(); //bir tane transaction oluşturur ve oluşturulan transaction içerisinde olması gereken update sorguları tek seferde işlenmiş olur.

**Veri Nasıl Silinir?**

ETicaretContext context = new();

Urun urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 4);

context.Urunler.Remove(urun); //4 id'sine sahip olan veriyi silmiş olduk

await context.SaveChangesAsync();

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

}

**Takip Edilmeyen Nesneler Nasıl Silinir?**

ETicaretContext context = new ETicaretContext();

Urun u = new Urun

{

Id = 2 //unique,primary key olan değeri yazılır.

};

context.Urunler.Remove(u);

await context.SaveChangesAsync();

**EntityState ile Silme İşlemi**

ETicaretContext context = new ETicaretContext();

Urun u = new() { Id = 3 };

context.Entry(u).State = EntityState.Deleted;

await context.SaveChangesAsync();

**RemoveRange ile Çoklu Silme**

ETicaretContext context = new();

List<Urun> urunler = await context.Urunler.Where(u => u.Id >= 7 && u.Id <= 9).ToListAsync(); //id'sine 7'den büyük eşit ve 9'dan küçük eşit verileri getir.

context.Urunler.RemoveRange(urunler); //id 7,8,9'a sahip ürünler silindi.

await context.SaveChangesAsync();

**DERS 5: Sorgu İşlemleri**

**En Temel Basit Bir Sorgulama Nasıl yapılır?**

using Microsoft.EntityFrameworkCore;

ETicaretContext context = new();

**#region En Temel Basit Bir Sorgulama Nasıl yapılır?**

**#region Method Syntax**

//Sorgulama sürecinde metotları kullanırsak buna Method Syntax denir.

var urunler = await context.Urunler.ToListAsync();

**#endregion**

**#region Query Syntax**

//ürünler tablosunun içerisindeki her bir ürünü bana getir.

var urunler2 = await (from urun in context.Urunler

//Urunler tablosunda yapılan bu sorgulamadaki her bir veriyi urun değişkeni ile (adı farklı olabilir) temsil edip select ile çek.

select urun).ToListAsync();

**#endregion**

**#endregion**

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

public DbSet<Parca> Parcalar { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

public ICollection<Parca> Parcalar { get; set; }

}

public class Parca

{

public int Id { get; set; }

public string ParcaAdi { get; set; }

}

**Not:**

IQueryable ve IEnumerable Nedir? Basit anlatım.

- IQueryable sorguya karşılık gelir. EF Core üzerinden yapılmış olan sorgunun execute edilmemiş halini ifade eder.

- IEnumerable sorgunun çalıştırılıp/execute edilip verilerin in memory'e yüklenmiş (artık veriler yazılımda entity instance'ları olarak tutuluyor) halini ifade eder.

IQueryable'dayız.

var urunler = from urun in context.Urunler

select urun;

IQueryable'dan IEnumerable'a getirmek için ToListAsync ekleriz.

var urunler = (from urun in context.Urunler

select urun).ToListAsync();

**Sorguyu Execute Etmek İçin Ne Yapmamız Gerekmektedir?**

//ToListAsync kullanabiliriz.

//ya da

var urunler = from urun in context.Urunler

select urun;

//foreach

foreach (Urun urun in urunler)

{

Console.WriteLine(urun.UrunAdi);

}

**Deferred Execution (Ertelenmiş Çalışma)**

- IQeryable çalışmalarında ilgili kod, yazıldığı anda tetiklenmez/çalıştırılmaz. Yani ilgili kod yazıldığı noktada sorguyu generate etmez!

- Çalıştırıldığı/execute edildiği noktada tetiklenir. İşte buna ertelenmiş çalışma denir.

Örnek:

int urunId = 5;

var urunler = from urun in context.Urunler

where urun.Id > urunId

select urun;

urunId = 200; //foreach'de execute edilirken 200 değerine göre çalışmayı gerçekleştirecektir.

//foreach

foreach (Urun urun in urunler)

{

Console.WriteLine(urun.UrunAdi);

}

Örnek:

int urunId = 5;

string urunAdi = "2";

var urunler = from urun in context.Urunler

where urun.Id > urunId && urun.UrunAdi.Contains(urunAdi)

//başka bir yerde urunAdi belirtmezsek, 2 barındıranları getirir.

select urun;

urunId = 200; //foreach'de execute edilirken 200 değerine göre çalışmayı gerçekleştirecektir.

urunAdi = "4"; //artık 4 olanları getirecektir.

//foreach

foreach (Urun urun in urunler)

{

Console.WriteLine(urun.UrunAdi);

}

Örnek:

int urunId = 5;

string urunAdi = "2";

var urunler = from urun in context.Urunler

where urun.Id > urunId && urun.UrunAdi.Contains(urunAdi)

//başka bir yerde urunAdi belirtmezsek, 2 barındıranları getirir.

select urun;

urunId = 200; //ToListAsync'te de execute edilirken 200 değerine göre çalışmayı gerçekleştirecektir.

urunAdi = "4"; //artık 4 olanları getirecektir.

await urunler.ToListAsync();

**Not:**

ETicaretContext context = new();

Artık using'in altında global olarak oluşturulmaktadır.

**Çoğul Veri Getiren Sorgulama Fonksiyonları**

**ToListAsync Fonksiyonu**

#region ToListAsync

//Üretilen sorguyu execute ettirmemizi sağlayan fonksiyondur.

//Method Syntax

var urunler = context.Urunler.ToListAsync();

//Query Syntax

var urunler = (from urun in context.Urunler

select urun).ToListAsync();

ya da

var urunler = (from urun in context.Urunler

select urun);

var datas = await urunler.ToListAsync();

#endregion

**Where Fonksiyonu**

#region Where

//Oluşturulan Sorguya Where şartı eklememizi sağlayan bir fonksiyondur.

//Method Syntax

var urunler = await context.Urunler.Where(u => u.Id > 500).ToListAsync();

var urunler2 = await context.Urunler.Where(u => u.UrunAdi.StartsWith("a")).ToListAsync();

var urunler3 = await context.Urunler.Where(u => u.UrunAdi.EndsWith("b")).ToListAsync();

var urunler4 = await context.Urunler.Where(u => u.UrunAdi.Contains("c")).ToListAsync();

//Query Syntax

var urunler5 = from urun in context.Urunler

where urun.Id > 500 && urun.UrunAdi.EndsWith("7")

select urun;

var data = await urunler5.ToListAsync();

#endregion

**OrderBy Fonksiyonu**

#region OrderBy

//Sorgu üzerinde sıralama yapmamızı sağlayan bir fonksiyondur.

//Ascending olarak sıralamayı yapar.

//Method Syntax

var urunler = context.Urunler.Where(u => u.Id > 500 || u.UrunAdi.StartsWith("2")).OrderBy(u => u.UrunAdi);

//Query Syntax

var urunler2 = from urun in context.Urunler

where urun.Id > 500 || urun.UrunAdi.EndsWith("2")

orderby urun.UrunAdi

select urun;

await urunler.ToListAsync();

await urunler2.ToListAsync();

#endregion

**ThenBy Fonksiyonu**

#region ThenBy

//OrderBy üzerinde yapılan sıralama işlemini farklı kolonlara da uygulamamızı sağlayan bir fonksiyondur. (Ascending)

//Method Syntax

var urunler = context.Urunler.Where(u => u.Id > 500 || u.UrunAdi.StartsWith("2")).OrderBy(u => u.UrunAdi).ThenBy(u => u.Fiyat).ThenBy(u => u.Id);

await urunler.ToListAsync();

#endregion

**OrderByDescending Fonksiyonu**

#region OrderByDescending

//Descending olarak sıralama yapmamızı sağlayan bir fonksiyondur.

//Method Syntax

var urunler = await context.Urunler.OrderByDescending(u => u.Fiyat).ToListAsync();

//Query Syntax

var urunler2 = await (from urun in context.Urunler

orderby urun.UrunAdi descending

select urun).ToListAsync();

#endregion

**ThenByDescending Fonksiyonu**

#region ThenByDescending

//OrderByDescending üzerinde yapılan sıralama işlemini farklı kolonlara da uygulamamızı sağlayan bir fonksiyondur. (Ascending)

var urunler = await context.Urunler.OrderByDescending(u => u.Id).ThenByDescending(u => u.Fiyat).ThenBy(u => u.UrunAdi).ToListAsync();

#endregion

**Tekil Veri Getiren Sorgulama Fonksiyonları**

- Single ya da SingleOrDefault Fonksiyonları: Oluşturduğumuz sorguda sadece tek bir sonucun gelmesini istiyorsak, birden fazla sonuç geliyorsa ya da hiç gelmiyorsa hata(exception) fırlatılmasını istiyorsak bu fonksiyonları kullanabiliriz. Ufak bir fark vardır.

Single ya da SingleAsync: Sorgu neticesinde birden fazla veri geliyorsa ya da hiç gelmiyorsa her iki durumda da exception fırlatır.

SingleOrDefault ya da SingleOrDefaultAsync: Sorgu neticesinde birden fazla veri geliyorsa exception fırlatır, hiç gelmiyorsa null döner.

**SingleAsync Kullanımı,**

Tek Bir Kayıt Geldiğinde,

var urun = await context.Urunler.SingleAsync(u => u.Id == 55);//sorun yok. Tek bir veri geliyor. Satırı döndürür.

Hiç Kayıt Gelmediğinde,

var urun = await context.Urunler.SingleAsync(u => u.Id == 5555); //5555 Id'li birisi yoksa Exception hatası fırlatır.

Birden Fazla Kayıt Geldiğinde,

var urun = await context.Urunler.SingleAsync(u => u.Id > 55); //Id'si 55'ten büyük birden fazla kayıt olsun.

//Yine Exception hatası alınacaktır.

**SingleOrDefaultAsync Kullanımı,**

Tek Bir Kayıt Geldiğinde,

var urun = await context.Urunler.SingleOrDefaultAsync(u => u.Id == 55); //sorun yok. Tek bir veri geliyor.Satırı döndürür.

Hiç Kayıt Gelmediğinde,

var urun = await context.Urunler.SingleOrDefaultAsync(u => u.Id == 5555); //5555 Id'li birisi yoksa Null döner.

Birden Fazla Kayıt Geldiğinde,

var urun = await context.Urunler.SingleOrDefaultAsync(u => u.Id > 55); //Id'si 55'ten büyük birden fazla kayıt olsun.

//Exception hatası alınacaktır.

**FirstAsync ve FirstOrDefaultAsync**

- Yapılan sorguda tek bir verinin gelmesi amaçlanıyorsa bu fonksiyonlar kullanılabilir.

- Eğer ki yapmış olduğumuz sorguda gelecek olan verilerin sadece tek bir veri olduğundan emin olmak istiyorsak, birden fazla veri geldiği durumda yazılımı uyarmak istiyorsak bu fonksiyonlar kullanılabilir.

**FirstAsync:** Sorgu neticesinde elde edilen verilerden ilkini getirir. Eğer ki hiç veri gelmiyorsa hata fırlatır.

Tek Kayıt Geldiğinde,

var urun = await context.Urunler.FirstAsync(u => u.Id == 55); //Hata yok. Satırı döndürür.

Hiç Kayıt Gelmediğinde,

var urun = await context.Urunler.FirstAsync(u => u.Id == 5555); //Exception.

Birden Fazla Kayıt Geldiğinde,

var urun = await context.Urunler.FirstAsync(u => u.Id > 55); //55'ten büyük Id'lere sahip verileri SQL'de elde edip, 55'ten sonra 56 Id'sine sahip olanı getirecektir.

**FirstOrDefaultAsync:** Sorgu neticesinde elde edilen verilerden ilkini getirir. Eğer ki hiç veri gelmiyorsa null değerini döndürür.

Tek Kayıt Geldiğinde,

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 55);//Hata yok. Satırı döndürür.

Hiç Kayıt Gelmediğinde,

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 5555); //Null döner.

Birden Fazla Kayıt Geldiğinde,

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id > 55);//FirstAsync ile aynı çalışmayı sergileyecektir. Birden fazla verinin ilkini döndürecektir.

Not: Single, First, SingleOrDefault, FirstOrDefault fonksiyonları, sorguyu her zaman veritabanına gönderir. Tüm kolonları where ile sorgulayabilir.

**FindAsync**

- Sorgulama sürecinde önce context içerisini(bellek, in memory) kontrol eder, kaydı bulamadığı taktirde sorguyu veritabanına gönderir. Yalnızca primary key alanlarını sorgulayabilir. Kayıt bulamazsa null döndürür.

Urun urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 55);

//Id primary key olduğundan aşağıdaki gibi Find'ı kullanabiliriz.

//Find fonksiyonu, primary key kolonuna özel hızlı bir şekilde sorgulama yapmamızı sağlayan bir fonksiyondur.

Urun urun = await context.Urunler.FindAsync(55);

//55 Id'sine sahip ürünü direkt olarak getirecektir.

**Find ile Composite Primary Key Durumu**

UrunParca u = await context.UrunParca.FindAsync(2,5);

Not:Veri tabanında tablo oluştururken, tablodaki kayıtların benzersiz olması için her tabloya genellikle bir tane birincil anahtar yerleştiririz. Ancak bazı durumlarda 1 tane birincil anahtar tek başına istediğimiz şekilde tablodaki sorunları çözmek için yeterli olmayabilir.

Birden fazla alanın birlikte birincil olması gerektiği durumlarda birincil anahtar olarak ayarlaması gerekir. Bu duruma composite key adı verilir.

**LastAsync ve LastOrDefaultAsync**

- Bu fonksiyonları OrderBy ile birlikte kullanmamız gereklidir.

var urun = await context.Urunler.OrderBy(u => u.UrunAdi).LastAsync(u => u.Id > 55);//UrunAdi'na göre sıralamayı yapıp sonuncusunun Id'sini getirecektir.

var urun = await context.Urunler.OrderBy(u => u.Fiyat).LastAsync(u => u.Id > 55); //Fiyat'a göre sıralamayı yapıp sonuncusunun Id'sini getirecektir.

LastAsync'de hiç veri gelmiyorsa hata fırlatır.

LastOrDefaultAsync'da hiç veri gelmiyorsa null döndürür.

var urun = await context.Urunler.OrderBy(u => u.Fiyat).LastOrDefaultAsync(u => u.Id > 55); //LastAsync ile aynı sonuç alınacaktır.

**Diğer Sorgulama Fonksiyonları**

**CountAsync**

- Oluşturulan sırgunun execute edilmesi neticesinde kaç adet satırın elde edileceğini sayısal olarak (int) bize bildiren fonksiyondur.

var urunler = (await context.Urunler.ToListAsync()).CountAsync(); //maliyetlidir. Çok sayıda ürünü belleğe çekip orada sayıyoruz.

var urunler = await context.Urunler.CountAsync(); //daha az maliyetli.

//Bu şekilde IQueryable'da iken sorgunun devamını Count ile getirmek direkt olarak veritabanından int değer döneceğinden maliyet olarak daha avantajlı olacaktır.

**LongCountAsync:** Oluşturulan sırgunun execute edilmesi neticesinde kaç adet satırın elde edileceğini sayısal olarak (long) bize bildiren fonksiyondur.

var urunler = await context.Urunler.LongCountAsync();

var urunler = await context.Urunler.LongCountAsync(u => u.Fiyat > 5000); //Count fonksiyonlarında şartlı sorgu da yapılabilir.

**AnyAsync:** Sorgu neticesinde verinin gelip gelmediğini bool türünde veren fonksiyondur.

var urunler = await context.Urunler.AnyAsync(); //veri geliyorsa true. Gelmiyorsa false.

var urunler = await context.Urunler.Where(u => u.UrunAdi.Contains("1")).AnyAsync(); //verinin içinde 1 geçiyor mu? True ya da False.

**MaxAsync:** Oluşturulan sorguda verilen kolonda sayısal olarak en yüksek olan değer hangisiyse onu getirir.

var fiyat = await context.Urunler.MaxAsync(u => u.Fiyat); //En yüksek Fiyat'ı getirir.

**MinAsync:** Oluşturulan sorguda verilen kolonda sayısal olarak en düşük olan değer hangisiyse onu getirir.

var fiyat = await context.Urunler.MinAsync(u => u.Fiyat); //En düşük Fiyat'ı getirir.

**Distinct:** Sorguda tekrar eden kayıtlar varsa bunları tekilleştiren bir işleve sahip fonksiyondur.

var urunler = await context.Urunler.Distinct().ToListAsync(); //IQueryable döndüğünden ToListAsync ile execute etmemiz gerekir ki veritabanına gönderebilelim.

**AllAsync:** Bir sorgu neticesinde gelen verilerin, veirlen şarta uyup uymadığını kontrol etmektedir. Eğer ki tüm veriler şarta uyuyorsa true, uymuyorsa false döndürecektir.

var m = await context.Urunler.AllAsync(u => u.Fiyat > 5000); //bütün veriler 5000'den büyük ise true, değilse false.

var m = await context.Urunler.AllAsync(u => u.Fiyat < 15000);

var m = await context.Urunler.AllAsync(u => u.UrunAdi.Contains("a));

**SumAsync:** Vermiş olduğumuz sayısal property'nin toplamını alır.

var fiyatToplam = await context.Urunler.SumAsync(u => u.Fiyat);

**AverageAsync**: Vermiş olduğumuz sayısal property'nin aritmetik ortalamasını verir.

var aritmetikOrtalama = await context.Urunler.AverageAsync(u => u.Fİyat);

**Contains**: İçinde geçen şeklinde, Like'%...%' sorgusu oluşturmamızı sağlar. Where şartının içerisinde belirtilmesi lazım.

EF Core, UrunAdi üzerinde Contains'i görürse bunun bir Like sorgusu olduğunu anlayacaktır. IQueryable'da olduğumuzdan ToListAsync ile execute etmeliyiz.

var urunler = await context.Urunler.Where(u => u.UrunAdi.Contains("7")).ToListAsync(); //7 içerenleri getir.

StarsWith: Like '...%' sorgusu oluşturmamızı sağlar.

var urunler = await context.Urunler.Where(u => u.UrunAdi.StarsWith("7")).ToListAsync(); //7 ile başlayanları getir.

EndsWith: Like '%...' sorgusu oluşturmamızı sağlar.

var urunler = await context.Urunler.Where(u => u.UrunAdi.EndsWith("7")).ToListAsync(); //7 ile bitenleri getir.

- StarsWith ve EndsWith'te de aynı kurallar geçerli. Where ile kullan. ToListAsync ekle.

**DERS 6: Sorgu Sonucu Dönüşüm Fonksiyonları**

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

ETicaretContext context = new();

Console.WriteLine("a");

- Bu fonksiyonlar ile sorgu neticesinde elde edilen verileri isteğimiz doğrultusunda farklı türlerde projeksiyon edebiliyoruz.

**ToDictionaryAsync**

- Sorgu neticesinde gelecek olan veriyi bir Dictionary olarak elde etmek/karşılamak istiyorsak eğer kullanabiliriz.

- ToList ile aynı amaca hizmet etmektedir. Yani oluşturulan sorguyu execute edip neticesini alırlar. Fakat ToList gelen sorgu neticesini Entity türünde bir koleksiyona (List<TEntity>) dönüştürmekteyken, ToDictionary ise gelen sorgu neticesini Dictionary türünden bir koleksiyona dönüştürecektir.

- Dictionary; (key,value) formatında verileri tutmamızı sağlayan bir koleksiyon çeşidididir.

var urunler = context.Urunler.ToDictionaryAsync(u => u.UrunAdi, u => u.Fiyat);

**ToArrayAsync**

- Oluşturulan sorguyu dizi olarak elde eder. ToList ile aynı amaca hizmet eder. Sorguyu execute eder lakin gelen sonucu entity dizisi olarak elde eder.

var urunler = context.Urunler.ToArrayAsync(); //üzerine sorgu çekmiş olduğumuz DbSet Urunler property'sine onun generic türüne uygun bir dizide sonuç döndürecektir.

- ToArray,ToList,ToDictionary'de de SQL kısmında aynı sorgu oluşturulur.

**Select**

- Select fonksiyonunun işlevsel olarak birden fazla davranışı söz konusudur.

1) Select fonksiyonu, generate edilecek sorgunun çekilecek kolonlarını ayarlamamızı sağlamaktadır.

ToList,ToArray,ToDictionary'den önce (IQueryable iken) kullanmamız lazım.

var urunler = context.Urunler.Select(u => new Urun

{

Id = u.Id,

Fiyat = u.Fiyat,

//sadece Id ve Fiyat kolonlarını getirir. Diğer kolonlar null olarak kalacaktır.

}).ToListAsync();

2) Select fonksiyonu, gelen verileri farklı türlerde karşılamamızı sağlar. T, anonim gibi.

var urunler2 = context.Urunler.Select(u => new

{

Id = u.Id,

Fiyat = u.Fiyat,

//bu şekilde de elde edilen veriler Anonymous Type olarak elde edilecektir.

}).ToListAsync();

var urunler3 = context.Urunler.Select(u => new UrunDetay

{

Id = u.Id,

Fiyat = u.Fiyat,

//bu şekilde de elde edilen veriler UrunDetay türünde elde edilecektir.

}).ToListAsync();

**SelectMany**

- Select ile aynı amaca hizmet eder. Lakin, ilişkisel tablolar neticesinde gelen koleksiyonel verileri de tekilleştirip projeksiyon etmemizi sağlar.

/\*

SELECT [u].[Id], [u].[Fiyat], [p].[ParcaAdi]

FROM [Urunler] AS [u]

INNER JOIN [Parcalar] AS [p] ON [u].[Id] = [p].[UrunId]

var urunler = await context.Urunler.Include(u => u.Parcalar).SelectMany(u => u.Parcalar, (u,p) => new

{

u.Id,

u.Fiyat,

p.ParcaAdi

}).ToListAsync();

**GroupBy Fonksiyonu**

- Gruplama yapmamızı sağlayan fonksiyondur.

/\*

SELECT Fiyat, COUNT(\*) From Urunler

GROUP By Fiyat

\*/

#region Method Syntax

var datas = await context.Urunler.GroupBy(u => u.Fiyat).Select(group => new

{

Count = group.Count(),

Fiyat = group.Key

}).ToListAsync();

// SQL Server Profiler'a bakarsak;

SELECT COUNT(\*) AS [Count], [u].[Fiyat]

FROM [Urunler] AS [u]

GROUP BY [u].[Fiyat]

#endregion

#region Query Syntax

var datas = await (from urun in context.Urunler

group urun by urun.Fiyat

into @group //gruplama işlemi neticesinde elde edilen veriyi group ile temsil et. @ verbatim ile keyword olmaktan çıkart.

select new

{

Fiyat = @group.Key,

Count = @group.Count()

}).ToListAsync();

/\*

SELECT [u].[Fiyat], COUNT(\*) AS [Count]

FROM [Urunler] AS [u]

GROUP BY [u].[Fiyat]

\*/

#endregion

**Foreach Fonksiyonu**

- Bir sorgulama fonksiyonu değildir.

- Sorgulama neticesinde elde edilen koleksiyonel veriler üzerinde iterasyonel olarak dönmemizi ve teker teker verileri elde edip işlemler yapabilmemizi sağlayan bir fonksiyondur.

- Foreach döngüsünün metot halidir.

var datas = await (from urun in context.Urunler

group urun by urun.Fiyat

into @group //gruplama işlemi neticesinde elde edilen veriyi group ile temsil et. @ verbatim ile keyword olmaktan çıkart.

select new

{

Fiyat = @group.Key,

Count = @group.Count()

}).ToListAsync();

foreach (var item in datas)

{

}

//ya da

datas.ForEach(x =>

{

});

//ortak kod alanı

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

public DbSet<Parca> Parcalar { get; set; }

public DbSet<UrunParca> UrunParcalari { get; set; }

public DbSet<UrunDetay> UrunDetaylari { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB2;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

public ICollection<Parca> Parcalar { get; set; }

}

public class Parca

{

public int Id { get; set; }

public string ParcaAdi { get; set; }

}

public class UrunParca

{

public int UrunId { get; set; }

public float ParcaId { get; set;}

public Urun Urun { get; set; }

public Parca Parca { get; set; }

}

public class UrunDetay

{

public int Id { get; set; }

public float Fiyat { get; set; }

}

**DERS 7: ChangeTracker Detaylı Anlatım**

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

ETicaretContext context = new();

Console.WriteLine("a");

**#region Change Tracking Nedir?**

//Context nesnesi üzerinden gelen tüm nesneler/veriler otomatik olarak bir takip mekanizması tarafından izlenirler.

//İşte bu takip mekanizmasına Change Tracker denir. Change Tracker ile nesneler üzerindeki değişiklikler/işlemler takip edilerek //netice itibariyle bu işlemlerin fıtratına uygun SQL sorgucukları generate edilir. İşte bu işlemene Change Tracking denir.

**#endregion**

**#region ChangeTracker Property'si**

//Takip edilen nesnelere erişelebilmemizi sağlayan ve gerektiği taktirde işlemler gerçekleştirmemizi sağlayan bir property'dir.

//Context sınıfının base class'ı olan DbContext sınıfının bir member'ıdır.

/\*

var urunler = await context.Urunler.ToListAsync();

var datas = context.ChangeTracker.Entries();

//datas içerisinde takip edilen bütün değerleri/nesneleri Unchanged olarak görebiliriz.

\*/

/\*

var urunler = await context.Urunler.ToListAsync();

urunler[6].Fiyat = 123; //update

context.Urunler.Remove(urunler[7]); //delete

urunler[8].UrunAdi = "Corap";

//index no 6 ile 8'ın State'leri Modified; 7.'nin Deleted olacaktır.

var datas = context.ChangeTracker.Entries(); //burada değişiklikler tutuluyor ki,

await context.SaveChangesAsync(); //çağırıldığı zaman ChangeTracker'da tutulan state'lere uygun sorgular generate edilebilsin //ve veritabanına göderilerek execute edilebilsin.

\*/

**#endregion**

**#region DetectChanges Metodu**

/\*

Yapılan operasyonlarda güncel tracking verilerinden emin olabilmek için bu fonksiyonu kullanabiliriz.

EF Core, context nesnesi tarafından izlenen tüm nesnelerdeki değişiklikleri Chan ge Tracker sayesinde takip edebilmekte ve nesnelerde var olan verisel değişiklikler

yakalanarak bunların anlık görüntüleri(snapshot)'ini oluşturabilir.

Yapılan değişikliklerin veritabanına gönderilmeden önce algılandığından emin olmak gerekir. SaveChanges fonksiyonu çağrıldığı anda nesneler EF Core tarafından otomatik kontrol edilirler. Ancak yapılan operasyonlarda güncel tracking verilerinden emin olabilmek için değişikliklerin algılanmasında opsiyonel olarak gerçekleştirmek isteyebiliriz. İşte bunun için DetectChanges fonksiyonu kullanılabilir ve her ne kadar EF Core değişiklikleri otomatik algılıyor olsa da biz yine de irademizle

kontrole zorlayabiliriz.

\*/

/\*

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 3);

urun.Fiyat = 123;

context.ChangeTracker.DetectChanges();

await context.SaveChangesAsync();

\*/

**#endregion**

**#region AutoDetectChangesEnabled Property'si**

/\*

İlgili metotlar(SaveChanges, Entries) tarafından DetectChanges metodu otomatik olarak tetiklenmesinin konfigürasyonunu yapmamızı sağlayan property'dir.

SaveChanges fonksiyonu tetiklendiğinde DetectChanges metodunu içerisinde default olarak çağırmaktadır. Bu durumda DetectChanges fonksiyonunun kullanımını irademizle

yönetmek ve maliyet/performans optimizasyonu yapmak istediğimiz durumlarda AutoDetectChangesEnabled özelliğini kapatabiliriz.

\*/

**#endregion**

**#region Entries Metodu**

/\*

Context'teki Entry metodunun koleksiyonel versiyonudur.

ChangeTracker mekanizması tarafından izlenen her entity nesnesinin bilgisini EntityEntry türünden elde etmemizi sağlar ve belirli işlemler yapabilmemize olanak tanır.

Entries metodu, DetectChanges metodunu tetikler. Bu durumda tıpka SaveChanges'da olduğu gibi bir maliyettir.

Buradaki maliyetten kaçınmak için AutoDetectChangesEnabled özelliğine false değeri verilebilir.

\*/

/\*

var urunler = await context.Urunler.ToListAsync();

urunler.FirstOrDefault(u => u.Id == 7).Fiyat = 123; //update

context.Urunler.Remove(urunler.FirstOrDefault(u=> u.Id == 8)); //delete

urunler.FirstOrDefault(u => u.Id == 9).UrunAdi = "Corrrrap"; //update

context.ChangeTracker.Entries().ToList().ForEach(e =>

{

if (e.State == EntityState.Unchanged)

{

//State'i Unchanged ise buradaki işlemleri uygula.

}

else if (e.State == EntityState.Deleted)

{

//State'i Deleted ise buradaki işlemleri uygula.

}

});

\*/

**#endregion**

**#region AcceptAllChanges Metodu**

/\*

SaveChanges() veya SaveChanges(true) tetiklendiğinde EF Core herşeyin yolunda olduğunu varsayarak track ettiği verilerin takibini keser ve yeni değişikliklerin takip edilmesini bekler. Böyle bir durumda beklenmeyen bir durum/olası bir hata söz konusu olursa eğer EF Core takip ettiği nesneleri bırakacağı için bir düzeltme metodu mevzu bahis olmayacaktır.

Haliyle bu durumda devreye SaveChanges(false) ve AcceptAllChanges metotları girecektir.

SaveChanges(false) EF Core'a gerekli veritabanı komutlarını yürütmesini söyler ancak gerektiğinde yeniden oynatılabilmesi için değişiklikleri beklemeye/nesneleri takip etmeye devam eder. Ta ki AcceptAllChanges metodunu irademizle çağırana kadar.

SaveChanges(false) ile işlemin başarılı olduğundan emin olursanız AcceptAllChanges metodu ile nesnelerden takibi kesebilirsiniz.

\*/

/\*

var urunler = await context.Urunler.ToListAsync();

urunler.FirstOrDefault(u => u.Id == 7).Fiyat = 123; //update

context.Urunler.Remove(urunler.FirstOrDefault(u => u.Id == 8)); //delete

urunler.FirstOrDefault(u => u.Id == 9).UrunAdi = "Corrrrap"; //update

//await context.SaveChangesAsync();

//await context.SaveChangesAsync(true);

await context.SaveChangesAsync(false);

context.ChangeTracker.AcceptAllChanges();

\*/

**#endregion**

**#region HasChanges Metodu**

//Takip edilen nesneler arasından değişiklik yapılanların olup olmadığının bilgisini verir. Arka planda DetectChanges metodunu tetikler.

//var result = context.ChangeTracker.HasChanges(); //sonuç olarak bool döner.

**#endregion**

**#region Entity States**

//Entity nesnelerinin durumlarını ifade eder.

#region Detached

/\*

//nesnenin change tracker mekanizması tarafından takip edilmediğini ifade eder.

Urun urun = new();

Console.WriteLine(context.Entry(urun).State); //detached yazacaktır. context üzerinden gelmiyor. ChangeTracker ile takip edilmiyor.

urun.UrunAdi = "Cururud";

await context.SaveChangesAsync();

\*/

**#endregion**

**#region Added**

//veritabanına eklenecek nesneyi ifade eder. Henüz veritabanına işlenmeyen veriyi ifade eder.

//SaveChanges fonksiyonu çağırıldığında insert sorgusu oluşturulacağı anlamına gelir.

/\*

Urun urun = new() { Fiyat = 123, UrunAdi = "Ürün 1001" };

Console.WriteLine(context.Entry(urun).State); //detached

await context.Urunler.AddAsync(urun); //şu andan itibaren takip söz konusu.

Console.WriteLine(context.Entry(urun).State); //added

await context.SaveChangesAsync();

urun.Fiyat = 312;

Console.WriteLine(context.Entry(urun).State); //Modified

await context.SaveChangesAsync();

\*/

#endregion

**#region Unchanged**

//Veritabanında sorgulandığında beri nesne üzerinde herhangi bir değişiklik yapılmadığını ifade eder. Sorgu neticesinde elde //edilen tüm nesneler başlangıçta bu state değerindedir.

//var urunler = await context.Urunler.ToArrayAsync();

//var data = context.ChangeTracker.Entries();

**#endregion**

**#region Modified**

//Nesne üzerinde değişiklik değişiklik ya da güncelleme yapıldığını ifade eder. SaveChanges fonksiyonu çağırıldığında update sorgusu oluşturulacağı anlamına gelir.

/\*

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 3);

Console.WriteLine(context.Entry(urun).State); //unchanged

urun.UrunAdi = "adibas";

Console.WriteLine(context.Entry(urun).State); //modified

await context.SaveChangesAsync();

Console.WriteLine(context.Entry(urun).State); //unchanged

\*/

/\*

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 3);

Console.WriteLine(context.Entry(urun).State); //unchanged

urun.UrunAdi = "adibas";

Console.WriteLine(context.Entry(urun).State); //modified

await context.SaveChangesAsync(false);

Console.WriteLine(context.Entry(urun).State); //modified çünkü false yazdığımızdan dolayı AcceptAllChanged fonksiyonu manuel çalışmayı beklediğinden dolayı takip edilen

//nesneleri bırakmamıştır. Daha hala modified olarak kalmaya devam ediyor.

\*/

**#endregion**

**#region Deleted**

//nesnenin silindiğini ifade eder. SaveChanges fonksiyonu çağırıldığı anda delete sorgusu oluşturulacağı anlamına gelir.

/\*

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 5);

context.Urunler.Remove(urun);

Console.WriteLine(context.Entry(urun).State);

context.SaveChangesAsync();

\*/

**#endregion**

**#endregion**

**#region Context Nesnesi Üzerinden Change Tracker**

/\*

context.ChangeTracker. ile o anda takip edilen bütün nesnelere

context.Entry. ile o anki nesneye.

\*/

/\*

var urun = await context.Urunler.FirstOrDefaultAsync(u => u.Id == 6);

urun.Fiyat = 123;

urun.UrunAdi = "Silgi"; //Modified | Update

\*/

**#region Entry Metodu**

**#region OriginalValues Propery'si**

/\*

var fiyat = context.Entry(urun).OriginalValues.GetValue<float>(nameof(Urun.Fiyat));

var urunAdi = context.Entry(urun).OriginalValues.GetValue<string>(nameof(Urun.UrunAdi));

\*/

**#endregion**

**#region CurrentValues Property'si**

//var urunAdi = context.Entry(urun).CurrentValues.GetValue<string>(nameof(Urun.UrunAdi));

//ilgili instance'ın veritabanındaki değil Heap'teki değerini getirir.

**#endregion**

**#region GetDatabaseValues Metodu**

//var \_urun = await context.Entry(urun).GetDatabaseValuesAsync(); //vermiş olduğumuz nesneye karışık veritabanından en güncel halini yine entity olarak elde etmemizi sağlar.

**#endregion**

**#endregion**

**#endregion**

**#region Change Tracker'ın Interceptor Olarak Kullanılması**

//public class ETicaretContext : DbContext 'a gidip override SaveChanges -> Cancellation -> TAB TAB yazalım.

**#endregion**

public class ETicaretContext : DbContext

{

public DbSet<Urun> Urunler { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB3;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

public override Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)

{

//Biz ne kadar SaveChanges tetiklersek burası tetiklenecek. Burası tetiklenirler şunları yapabiliriz.

//veritabanına gönderilmeden önce araya girip operasyon gerçekleştirmek istiyorsak bu şekilde SaveChanges'ı override //edebiliriz.

var entries = ChangeTracker.Entries(); //entry'leri elde edip,

foreach (var entry in entries)//gerçek base class'taki SaveChanges tetiklenmeden önce foreach ile buradaki entry'lere girip

{

if (entry.State == EntityState.Added) //buradaki entry'lerin durumlarına göre belirli operasyonları gerçekleştirebiliriz.

{

//mesela ekleme işlemi yapılıyorsa entry.Entity... Entity üzerinden entry'e eriş ve işlemlere tabi tut şeklinde operasyonlar yapılabilir.

}

}

return base.SaveChangesAsync(cancellationToken);//base class'taki SaveChanges

}

}

public class Urun

{

public int Id { get; set; }

public string UrunAdi { get; set; }

public float Fiyat { get; set; }

}

**ChangeTracker'ın Davranışlarını Yönetmemizi Sağlayan Metotlar**

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

ETicaretContext context = new();

**#region AsNoTracking Metodu**

/\*

Context üzerinden gelen tüm datalar ChangeTracker üzerinden takip edilmektedir.

ChangeTracker, takip ettiği nesnelerin sayısıyla doğru orantılı olacak şekilde bir maliyete sahiptir.

O yüzden üzerinde işlem yapılmayacak verilerin takip edilmesi bizlere lüzumsuz yere bir maliyet çıkaracaktır.

AsNoTracking metodu, context üzerinden sorgu neticesinde gelecek olan verilerin ChangeTracker tarafından takip

edilmesini engeller.

AsNoTracking metodu ile ChangeTracker'ın ihtiyaç olmayan verilerdeki maliyetini törpülemiş oluruz.

AsNoTracking fonksiyonu ile yapılan sorgulamalarda, verileri elde edebililir, bu verileri istenilen noktalarda

kullanabilir lakin veriler üzerinde herhangi bir değişiklik/update işlemi yapamayız. Çünkü SaveChanges çağırıldığında

bu değişiklikleri yakalamış olan bir ChangeTracker mekanizmamız yok.

\*/

/\*

//var kullanicilar = await context.Kullanicilar.ToListAsync(); maliyetli ve düşük performanslı

var kullanicilar = await context.Kullanicilar.AsNoTracking().ToListAsync(); //daha verimli

foreach (var kullanici in kullanicilar)

{

Console.WriteLine(kullanici.Adi);

kullanici.Adi = $"yeni-{kullanici.Adi}"; //güncelleme yapılmayacaktır. Veriler ilk hali gibi kalacaktır.

//illa ki güncelleme yapmak istiyorsak update fonksiyonu üzerinden güncelleme yapabiliriz.

context.Kullanicilar.Update(kullanici); //context üzerinden Kullanicilar'a gidip ardından kullanici nesnesini

//Update fonksiyonuna verirsek manuel güncelleme yapabiliriz.

}

await context.SaveChangesAsync();

\*/

**#endregion**

**#region AsNoTrackingWithIdentityResolution**

/\*

ChangeTracker mekanizması sayesinde yinelenen datalar aynı instance'ları kullanırlar. Mesela iki kullanıcı'ya

admin rolü verilecekse tek admin nesnesi üzerinden veriliyor. AsNoTracking durumunda ise iki tane tekrar eden

admin nesnesi oluşturulup kullanıcılara admin rolü veriliyor. AsNoTracking ile yapılan sorgularda yinelenen

datalar farklı instance'larda karşılanırlar. Çünkü ChangeTracker ile bağımız koptu.

AsNoTrackingWithIdentityResolution: ChangeTracker mekanizması yinelenen veerileri tekil instance olarak getirir.

Burada ekstradan performans kazancı söz konusudur.

Bizler yaptığımız sorgularda takip mekanizmasının AsNoTracking metodu ile maliyetini kırmak isterken bazen

maliyete sebebiyet verebiliriz. Özellikle ilişkisel tabloları sorgularken bu duruma dikkat etmemiz gerekiyor.

AsNoTracking ile elde edilen veriler takip edilmeyeceğinden dolayı yinelenen verilerin ayrı instancelarda

olmasına sebebiyet veriyoruz. Çünkü ChangeTracker mekanizması takip ettiği nesneden bellekte varsa eğer

aynı nesneden bir daha oluşturma gereği duymaksızın o nesneye ayrı noktalardaki ihtiyacı gidermektedir.

Böyle bir durumda hem takip mekanizmasının maliyetini ortadan kaldırmak hem de yinelenen dataları tek bir

instance üzerinde karşılamak için AsNoTrackingWithIdentityResolution fonksiyonunu kullanabiliriz.

\*/

/\*

//var kullanicilar = await context.Kullanicilar.Include(k => k.Roller).ToListAsync(); //8 sonuç

//var kullanicilar = await context.Kullanicilar.Include(k => k.Roller).AsNoTracking().ToListAsync(); //10 sonuç

var kullanicilar = await context.Kullanicilar.Include(k => k.Roller).AsNoTrackingWithIdentityResolution().ToListAsync(); //9 sonuç

//yine 8 sonuç. En optimum çalışma.

//AsNoTrackingWithIdentityResolution fonksiyonu AsNoTracking fonksiyonuna nazaran görece yavaştır/maliyetlidir

//likin ChangeTracker'a nazaran daha performanslı ve az maliyetlidir.

\*/

**#endregion**

**#region AsTracking**

/\*Context üzerinden gelen dataların ChangeTracker tarafından takip edilmesini iradeli bir şekilde ifade etmemizi

sağlayan fonksiyondur.

Bir sonraki inceleyeceğimiz UseQueryTrackingBehavior metodunun davranışı gereği uygulama

seviyesinde ChangeTracker'ın default olarak devrede olup olmamasını ayarlıyor olacağız. Eğer ki default olarak

pasif hale getilirse böyle durumlarda takip mekanizmasının ihtiyaç olduğu sorgularda AsTracking fonksiyonunu

kullanabilir ve böylece takip mekanizmasını iradeli bir şekilde devreye sokmuş oluruz.

\*/

//var kitaplar = await context.Kitaplar.AsTracking().ToListAsync();

**#endregion**

**#region UseQueryTrackingBehavior**

//EF Core/uygulama seviyesinde ilgili context'ten gelen verilerin üzerinde ChangeTracker mekanizmasının

//davranışı temel seviyede belirlememizi sağlayan fonksiyondur. Yani konfigürasyon fonksiyonudur.

//public class ETicaretContext : DbContext'deki OnConfiguring fonksiyonunda ilgili context'in davranış modellemesini

//yapıyorduk.

**#endregion**

public class ETicaretContext : DbContext

{

public DbSet<Kullanici> Kullanicilar { get; set; }

public DbSet<Rol> Roller { get; set; }

public DbSet<Kitap> Kitaplar { get; set; }

public DbSet<Yazar> Yazarlar { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ETicaretDB3;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking); //takip etme

//Context üzerinden gelen datalar artık default olarak takip edilmeyeceğinden dolayı takip edilmesini

//istediğimiz noktalarda AsTracking fonksiyonunu kullananmamız gerekecektir.

//optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTrackingWithIdentityResolution); //takip etme ama yinelenen dataların ayrı instance'larda olmasını engelle

//optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.TrackAll); //default halidir. Takip edilmesini söyler.

}

}

public class Kullanici

{

public Kullanici() => Console.WriteLine("Kullanici nesnesi oluşturuldu.");

public int Id { get; set; }

public string Adi { get; set; }

public float Email { get; set; }

public string Password { get; set; }

public ICollection<Rol> Roller { get; set; }

}

public class Rol

{

public Rol() => Console.WriteLine("Rol nesnesi oluşturuldu.");

public int Id { get; set; }

public string RolAdi { get; set; }

public ICollection<Kullanici> Kullanicilar { get; set; }

}

public class Kitap

{

public Kitap() => Console.WriteLine("Kitap nesnesi oluşturuldu.");

public int Id { get; set; }

public string KitapAdi { get; set; }

public int SayfaSayisi { get; set; }

public ICollection<Yazar> Yazarlar { get; set; }

}

public class Yazar

{

public Yazar() => Console.WriteLine("Yazar nesnesi oluşturuldu.");

public int Id { get; set; }

public string YazarAdi { get; set; }

public ICollection<Kitap> Kitaplar { get; set; }

}

**Ders 8: EF Core'da İlişkisel Yapılar**

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

Console.WriteLine("a");

**#region Relationships(İlişkiler) Terimleri**

**#region Principal Entity(Asıl Entity)**

//Kendi başına var olabilen tabloyu modelleyen entity'e denir. Departmanlar tablosunu modelleyen 'Departman' entity'sidir.

**#endregion**

**#region Dependent Entity (Bağımlı Entity)**

//Kendi başına var olamayan, bir başka tabloya ilişkisel olarak bağımlı olan tabloyu modelleyen entity'e denir.

//Çalışanlar tablosunu modelleyen 'Calisan' entity'sidir.

**#endregion**

**#region Foreign Key**

//Principle Entity ile Dependent Entity arasındaki ilişkiyi sağlayan key'dir.

//Dependent Entity'de tanımlanır.

//Principal Entity'deki Principal Key'i tutar.

//Calisanlar'daki DepartmanId, Departmanlar'daki Id'ye karşılık geliyor.

**#endregion**

**#region Principal Key**

//Principal Entity'deki id'nin kendisidir. Principal Entity'nin kimliği olan kolonu ifade eden property'dir.

**#endregion**

**#region Navigation Property Nedir?**

//İlişkisel tablolar arasındaki fiziksel erişimi entity class'ları üzerinden sağlayan property'lerdir.

//Bir propert'nin navigation property olabilmesi için kesinlikle entity türünden olması gerekiyor.

class Calisan //Dependent Entity

{

public int Id { get; set; }

public string CalisanAdi { get; set; }

public int DepartmanId { get; set; } //ForeignKey'e karşılık geliyor.

public Departman Departman { get; set; } //Calisan entity'si açısından Departman'lara olan Navigation Property'miz. Tekli bir ilişki var. Her bir Calisan'ın bir Departman'ı var.

}

//Burada bire çok bir ilişki var. Çalışanların 1 tane departmanı var. Departmanların birden fazla çalışanları var.

class Departman //Principal Entity

{

public int Id { get; set; } //Principal Key

public string DepartmanAdi { get; set; }

public ICollection<Calisan> Calisanlar { get; set; }

//Departman'ların birden çok çalışanı var. Yani çoklu ilişki. Bunları da bir Departman için temsil etmek istersek ICollection<Calisan> List türünde Calisan isminde bir

//koleksiyon oluşturuyoruz. Departman entity'si açısından Calisan'lara olan Navigation Property'miz.

}

//Navigation property'ler entity'lerdeki tanımlana göre n'e n yahut 1'en şeklinde ilişki türlerini ifade etmektedirler. İleride detaylı!

**#endregion**

**#endregion**

**#region İlişki Nedir?**

**#region One to One**

//Çalışan ile adresi arasındaki ilişki,

//Eşler arasındaki ilişki gibi.

//Yukarıda eğer Departman'da Calisan'a tekil bir navigation barındırsaydı,

/\*

class Departman //Principal Entity

{

public int Id { get; set; } //Principal Key

public string DepartmanAdi { get; set; }

public Calisan Calisanlar { get; set; } //ilişki birebir olacaktı.

}

\*/

**#endregion**

**#region One to Many**

//Yukarıdaki örnek on to many'e örnektir. Calisan ile Departman arasındaki ilişki.

//Anne-çocuk ilişkisi gibi.

#endregion

**#region Many to many**

//Kardeşler arasındaki ilişki gibi.

//Çalışanlar ile projeler arasındaki ilişki.

**#endregion**

**#endregion**

**#region Entity Framework Core'da İlişki Yapılandırma Yöntemleri**

**#region Default Conventions**

//Varsayılan entity kurallarını kullanarak yapılan ilişki yapılandırma yöntemidir.

//Yukarıda yaptığımız Id'yi otomatik tanıma, Foreign Key'i isminden tanıma vs. Entity'lerin temel kurallarını kullanıyoruz.

//Navigation Property'leri kullanarak ilişki şablonlarını çıkarmaktadır.

**#endregion**

**#region Data Annotations Attributes**

//Entity'nin niteliklerine göre ince ayarlar yapmamızı sağlayan attribute'lardır.

// [Key], [ForeignKey]

**#endregion**

**#region FluentAPI**

//Entity modellerindeki ilişkileri yapılandırırken daha detaylı çalışmamızı sağlayan öntemdir.

**#endregion**

**#region HasOne**

//İlgili entity'nin ilişkisel entity'e birebir ya da bire çok olacak şekilde ilişkisini yapılandırmaya başlayan metottur.

**#endregion**

**#region HasMany**

//İlgili entity'nin ilişkisel entity'e çoka bir ya da çoka çok olacak ilişkisini yapılandırmaya başlayan metottur.

**#endregion**

**#region WithOne**

//HasOne ya da HasMany'den sonra birebir ya da çoka bir olacak şekilde ilişki yapılandırmasını tamamlayan metottur.

**#endregion**

**#region WithMany**

//HasOne ya da HasMany'den sonra bire çok ya da çoka çok olacak şekilde ilişki yapılandırmasını tamamlayan metottur.

**#endregion**

**#endregion**

**EF Core'da Detaylı İlişkisel Yapılar**

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using System.Runtime.ConstrainedExecution;

ESirketDbContext context = new();

**#region Default Convention**

/\*

Her iki entity'de Navigation Property ile birbirlerini tekil olarak referans ederek fiziksel bir ilişkinin olacağı ifade edilir.

One to One ilişki türünde dependent entity'nin hangisi olduğunu default olarak belirleyebilmek pek kolay değildir. Bu durumda fiizksek olarak bir foreign key'e karşılık property/kolon tanımlayarak çözüm getirebiliyoruz. Böylece foreign key' karşılık property tanımlayarak lüzumsuz bir kolon oluşturmuş oluyoruz.

\*/

/\*

class Calisan

{

//Normal kolonlara karşılık gelen sıradan çinko karbon property'ler

public int Id { get; set; }

public string Adi { get; set; }

//Eğer ilgili property'nin türü Entity'nin türünden ise Navigation property.

public CalisanAdresi CalisanAdresi { get; set; }

}

class CalisanAdresi

{

//CalisanAdresi'nin dependent olduğunu EF Core bildirmek istiyorsak.

public int Id { get; set; } //bu bizim normal, primary key Id'miz.

public int CalisanId { get; set; }

//bu da bizim foreign key Id'miz olur. Artık CalisanAdresi dependent, Calisan ise Principal taraf olarak EF Core tarafından //algılanacak.

//bu default kurallar gereğidir.

//CalisanId artık Calisanlar Tablosundaki Id ile ilişkilendirildi.

public string Adres { get; set; }

public Calisan Calisan { get; set; }

}

\*/

**#endregion**

**#region Data Annotations**

/\*

Navigation Property'ler tanımlanmalıdır.

Foreign Key kolonunun ismi default convention dışında bir kolon olacaksa eğer ForeignKey attribute'u ile bunu bildirebiliriz.

Foreign Key kolonu oluşturulmak zorunda değildir.

1'e 1 ilişkide ekstradan foreign key kolonuna ihtiyaç olmacağından olayı dependent entity'deki ıd kolonunu hem foreign hem de primary key olarak kullanmayı tercih ediyoruz.

\*/

/\*

class Calisan

{

public int Id { get; set; }

public string Adi { get; set; }

public CalisanAdresi CalisanAdresi { get; set; }

}

class CalisanAdresi

{

[Key, ForeignKey(nameof(Calisan))]

public int Id { get; set; } //Id'yi hem primary key, hem de Foreign key atayarak tanımlayıp böylece index tanımlamadan unique olmuş oluyor ve

//ekstradan Foreign Key kolonu oluşturulmasına ihtiyaç kalmıyor. Maliyetten düşülüyor.

//iki tablo arasındaki ilişkiyi birebir garantisine almış olduk. Calisan ve CalisanId kesinlikle aynı değere sahip olacak. Ve primary key olduğundan dolayı da unique olacaktır.

//[ForeignKey(nameof(Calisan))]

//public int Cevdet { get; set; } //CalisanAdresleri içerisindeki Cevdet, Calisanlar'daki Id ile ilişikili.

//İsmini CalisanId'den değiştiğimiz an Data Annotations'a girmiş olduk. Fakat hala gereksiz bir kolon oluşturuluyor. Ve işe index'ler karışmış oluyor. Maliyetli!

//public string Adres { get; set; }

//public Calisan Calisan { get; set; }

}

\*/

**#endregion**

**#region FluentAPI**

/\*

Navigation Property'ler tanımlanmalıdır.

FluentAPI yönteminde Entity'ler arasındaki ilişki context sınıfı içerisinde OnModelCreating fonksiyonun override edilerek metotlar aracılığıyla tasarlanması gerekmektedir. Tüm Sorumluluk bu fonksiyon içerisindeki çalışmalardadır.

\*/

class Calisan

{

public int Id { get; set; }

public string Adi { get; set; }

public CalisanAdresi CalisanAdresi { get; set; }

}

class CalisanAdresi

{

public int Id { get; set; }

public string Adres { get; set; }

public Calisan Calisan { get; set; }

}

#endregion

class ESirketDbContext : DbContext

{

public DbSet<Calisan> Calisanlar { get; set; }

public DbSet<CalisanAdresi> CalisanAdresleri { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ESirketDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

//Model'ların(entity) veritabanında generate edilecek yapıları bu fonksiyonda konfigüre edilir.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//modelBuilder parametresi üzerinden Calisan'a gidelim. Calisan'dan HasOne ile CalisanAdresi ile birebir ilişki kuralım.

modelBuilder.Entity<Calisan>()

.HasOne(c => c.CalisanAdresi) //buradaki c, Calisan'ı temsil ediyor. Calisan'ın içerisindeki CalisanAdresi property'si bizim //navigation property'miz.HasOne diyerek bu navigation property'nin gösterdiği tabloya birebir ilişki kur demiş olduk.

.WithOne(c => c.Calisan) //buradaki c, CalisanAdresi'nin referansı(temsili). Bu da CalisanAdresi içerisindeki Calisan adresi //ile birebir ilişki kuran np'miz.

.HasForeignKey<CalisanAdresi>(c => c.Id); //Foreign Key yapısını tanıtıp hangi Entity'nin dependent olduğunu belirtiyoruz. //CalisanAdresi içerisindeki Id'nin foreign key olacağını bildiriyoruz. Fakat burada Primary Key özelliği ezildi. Aşağıda;

modelBuilder.Entity<CalisanAdresi>() //CalisanAdresi'ndeki Id'nin primary key olduğunu bildiriyoruz. Artık hem primary key hem foreign key.

.HasKey(c => c.Id);

}

//işte bu API metotları üzerinden olan çalışmaya FluentAPI deiyoruz.

}

**Bire Çok İlişki Türü**

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations.Schema;

Console.WriteLine("aaa");

**#region Default Convention**

/\*

Default Convention yönetminde bire çok ilişkiyi kurarken foreign key kolonuna karşılık gelen bir property tanımlamak mecburiyetinde değiliz. Eğer tanımlamazsak EF Core bunu

kendisi tamamlayacak yok eğer tanımlarsak, tanımladığımızı baz alacaktır.

\*/

/\*

class Calisan //Dependent Entity

{

public int Id { get; set; }

public string Adi { get; set; }

//public int DepartmanId { get; set; } yazılmış gibi SQL tarafında kendisi bir DepartmanId kolonu oluşturulacaktır.

public Departman Departman { get; set; }

}

class Departman

{

public int Id { get; set; }

public int DepartmanAdi { get; set;}

public ICollection<Calisan> Calisanlar { get; set; }

}

\*/

**#endregion**

**#region Data Annonations**

/\*

Default Convention yönteminde foreign key kolonuna karşılık gelen property'i tanımladığmızda bu property ismi temel geleneksel entity tanımlama kurallarına uymuyorsa eğer

Data Annotations'lar ile müdahelede bulunabiliriz.

\*/

/\*

class Calisan //Dependent Entity

{

public int Id { get; set; }

[ForeignKey(nameof(Departman))]

public int DId { get; set; }

public string Adi { get; set; }

public Departman Departman { get; set; }

}

class Departman

{

public int Id { get; set; }

public int DepartmanAdi { get; set; }

public ICollection<Calisan> Calisanlar { get; set; }

}

\*/

**#endregion**

**#region FluentAPI**

class Calisan //Dependent Entity

{

public int Id { get; set; }

public int DId { get; set; }

public string Adi { get; set; }

public Departman Departman { get; set; }

}

class Departman

{

public int Id { get; set; }

public int DepartmanAdi { get; set; }

public ICollection<Calisan> Calisanlar { get; set; }

}

**#endregion**

class ESirketDbContext : DbContext

{

public DbSet<Calisan> Calisanlar { get; set; }

public DbSet<Departman> Departmanlar { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ESirketDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

/\* Bu şekilde kendisi otomatik bir DepartmanId foreign key kolonu oluşturacaktır.

modelBuilder.Entity<Calisan>()

.HasOne(c => c.Departman)

.WithMany(d => d.Calisanlar);

\*/

//farklı isimde bir kolon oluşturmak istersek

modelBuilder.Entity<Calisan>()

.HasOne(c => c.Departman)

.WithMany(d => d.Calisanlar)

.HasForeignKey(e => e.DId);

}

}

**Çoka Çok İlişki Türü**

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Internal;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using System.Security.Cryptography;

Console.WriteLine("aaa");

**#region Default Convention**

/\*

İki entity arasındaki ilişkiyi Navigation property'ler üzerinden çoğul (ICollection ya da List vs.) olarak kurmalıyız.

Default Convention'da Cross Table'ı manuel oluşturmak zorunda değiliz. EF Core tasarıma uygun bir şekilde cross table'ı kendisi otomatik basacak ve generate edecektir.

Ve oluşturulan cross table'ın içerisinde composite primary key'i de otomatik oluşturmuş olacaktır.

class Kitap

{

public int Id { get; set; }

public string KitapAdi { get; set; }

public ICollection<Yazar> Yazarlar { get; set; }

}

class Yazar

{

public int Id { get; set; }

public int YazarAdi { get; set; }

public ICollection<Kitap> Kitaplar { get; set; }

}

\*/

**#endregion**

**#region Data Annotations**

/\*

//Cross table manuel olarak oluşturulmak zorundadır.

//Entity'leri oluşturduğumuz cross table entity'si ile bire çok bir ilişki kurulmalıdır.

//Cross table'da composite primary key'i data annotations(Attributes'lar) ile manuel kuramıyoruz. Bunun için de Fluent API'da çalışma yapmamız gerekiyor.

//Cross table'a karşılık bir entity modeli oluşturuyorsak eğer bunu Context sınıfı içerisinde DbSet property'si olarak bildirmek mecburiyetinde değiliz.

class Kitap

{

public int Id { get; set; }

public string KitapAdi { get; set; }

public ICollection<KitapYazar> Yazarlar { get; set; }

}

class KitapYazar

{

Composite primary key oluşturabilmek için iki tane [Key] yazmamıza Data Annotations'ları kullanırken izin yok. FLuentAPI'ya uğramak zorundayız.

[Key]

public int KitapId { get; set; }

[Key]

public int YazarId { get; set; }

Eğer ki KId ve YId olarak isimlendirme kuralları dışında çalışsaydık, yine KId ve YId'yi primary key olarak OnModelCreating'de atardı. Fakat,

Foreign Key olarak atamaz, onların yerin oromatik olarak KitapId ve YazarId oluşturur onları foreign key atardı. Bunu düzeltmek için,

[ForeignKey(nameof(Kitap))] //şeklinde tasarlamamız gerekirdi.

public int KId { get; set; }

[ForeignKey(nameof(Yazar))]

public int YId { get; set; }

public Kitap Kitap { get; set; }

public Yazar Yazar { get; set; }

--------------

public int KitapId { get; set; }

public int YazarId { get; set; }

public Kitap Kitap { get; set; }

public Yazar Yazar { get; set; }

}

class Yazar

{

public int Id { get; set; }

public int YazarAdi { get; set; }

public ICollection<KitapYazar> Kitaplar { get; set; }

}

class EKitapDbContext : DbContext

{

public DbSet<Kitap> Kitaplar { get; set; }

public DbSet<Yazar> Yazarlar { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=EKitapDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//hem KitapId hem YazarId'ye primary key verebilmek için new ile bir anonim tür oluşturduk. HasKey ile de manuel bir şekilde primary key olarak atadık.

//Ayrıca temel isimlendirme kuralları gereği EF Core KitapId ve YazarId'yi foreign key olarak da atayacaktır.

modelBuilder.Entity<KitapYazar>()

.HasKey(ky => new { ky.KitapId, ky.YazarId });

}

}

\*/

**#endregion**

**#region FluentAPI**

//Cross table manuel oluşturulmalıdır.

//DbSet olarak eklenmesine lüzum yoktur.

//Composite Primary Key HasKey metodu ile kurulmalı.

class Kitap

{

public int Id { get; set; }

public string KitapAdi { get; set; }

public ICollection<KitapYazar> Yazarlar { get; set; }

}

//Cross Table

class KitapYazar

{

public int KitapId { get; set; }

public int YazarId { get; set; }

public Kitap Kitap { get; set; }

public Yazar Yazar { get; set; }

}

class Yazar

{

public int Id { get; set; }

public int YazarAdi { get; set; }

public ICollection<KitapYazar> Kitaplar { get; set; }

}

class EKitapDbContext : DbContext

{

public DbSet<Kitap> Kitaplar { get; set; }

public DbSet<Yazar> Yazarlar { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=EKitapDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<KitapYazar>()

.HasKey(ky => new { ky.KitapId, ky.YazarId });

modelBuilder.Entity<KitapYazar>()

.HasOne(ky => ky.Kitap)

.WithMany(k => k.Yazarlar)

.HasForeignKey(ky => ky.KitapId);

modelBuilder.Entity<KitapYazar>()

.HasOne(ky => ky.Yazar)

.WithMany(k => k.Kitaplar)

.HasForeignKey(ky => ky.YazarId);

}

}

**#endregion**

**DERS 9: İlişkisel Senaryolarda Veri Ekleme**

using Microsoft.EntityFrameworkCore;

using System.Net;

using System;

using System.Net.Sockets;

Console.WriteLine("a");

ApplicationDbContext context = new();

**#region One to One İlişkisel Senaryolarda Veri Ekleme**

/\*

**#region 1. Yöntem : Principal Entity Üzerinden Dependenty Entity Verisi Ekleme**

//Eğer ki principal entity üzerinden ekleme gerçekleştiriliyorsa dependent entity nesnesi verilmek zorunda değildir!

//Fakat, dependent entity üzerinden ekleme işlemi gerçekleştiriliyorsa eğer burada principal entity'nin nesnesine ihtiyacımız //zaruridir.

Person person = new();

person.Name = "Faruk";

person.Address = new() { PersonAddress = "Çinçin/ANKARA" };

await context.AddAsync(person);

await context.SaveChangesAsync();

**#endregion**

**#region 2. Yöntem : Dependent Entity Üzerinden Principal Entity Verisi Ekleme**

Address address = new()

{

PersonAddress = "Yenişehir/Mersin",

Person = new() { Name = "Hakan"}

};

await context.AddAsync(address);

await context.SaveChangesAsync();

**#endregion**

class Person

{

public int Id { get; set; }

public string Name { get; set; }

public Address Address { get; set; }

}

class Address

{

public int Id { get; set; }

public string PersonAddress { get; set; }

public Person Person { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Person> Persons { get; set; }

public DbSet<Address> Addresses { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ApplicationDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating (ModelBuilder modelBuilder)

{

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

.HasForeignKey<Address>(a => a.Id);

}

}

\*/

**#endregion**

**#region One to Many İlişkisel Senaryolarda Veri Ekleme**

**#region 1. Yöntem: Principal Entity Üzerinden Dependent Entity Verisi Ekleme**

/\*

//Nesne Referansı Üzerinden Ekleme

Blog blog = new() { Name = "yazilimciboy33.com Blog" }; //Nesne referansı üzerinden ilgili veri ile ilişkisel dependent entity'leri,

blog.Posts.Add(new() { Title = "Post 1" }); //bu şekilde ekleyebilmek istiyorsak eğer,

blog.Posts.Add(new() { Title = "Post 2" }); //Post'un null olmaması lazım. (hata verir)

blog.Posts.Add(new() { Title = "Post 3" }); //Bu Post'un Null olmaması içinde bir yerde nesnesinin üretilmiş olması lazım. (aşağıda constructor içerisinde)

await context.AddAsync(blog);

await context.SaveChangesAsync();

//Object Initializers Üzerinden Ekleme

Blog blog2 = new()

{

Name = "A Blog",

Posts = new HashSet<Post>()

{

new() { Title ="Post 4" },

new() { Title ="Post 5" },

new() { Title ="Post 6" }

}

};

await context.AddAsync(blog2);

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 2. Yöntem: Dependent Entity Üzerinden Principal Entity Verisi Ekleme**

//Bire çok veri eklemede pek tercih edilen bir yöntem değildir.

/\*

Post post = new()

{

Title = "Post 42",

Blog = new() { Name = "B Blog"}

};

await context.AddAsync(post);

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 3. Yöntem: Foreign Key Kolonu Üzerinden Veri Ekleme**

//1. ve 2. yöntemler hiç veri olmadığı durumlarda ilişkisel veri eklememizi sağlayan yöntemlerken,

//bu 3. yöntem ise önceden eklenmiş olan bir principal entity verisiyle yeni dependent entity'lerin ilişkisel olarak eşleştirilmesini //sağlamaktadır.

/\*

Post post = new Post()

{

BlogId = 1,

Title = "Post 44"

};

await context.AddAsync(post);

await context.SaveChangesAsync();

\*/

#endregion

/\*

class Blog

{

public Blog()

{

Posts = new HashSet<Post>(); //Blog'un constructor'u içerisinde bu değeri, bir tane koleksiyonel değer olarak veriyoruz.

//List yerine HashSet kullanılmasının sebebi daha performanslı olması.

}

public int Id { get; set; }

public string Name { get; set; }

public ICollection<Post> Posts { get; set; } // Buradaki ICollection navigation property'sine bir değer vermemiz lazım.

}

class Post

{

public int Id { get; set; }

public int BlogId { get; set; }

public string Title { get; set; }

public Blog Blog { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ApplicationDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

\*/

**#endregion**

**#region Many to Many İlişkisel Senaryolarda Veri Ekleme**

**//1.Yöntem: n to n ilişkisi eğer ki default convention üzerinden tasarlanmışsa kullanılan bir yöntemdir.**

/\*

Book book = new()

{

BookName = "A Kitabı",

Authors = new HashSet<Author>()

{

new() { AuthorName = "Tolsoyewski" },

new() { AuthorName = "Necip Fazıl Asabıyanık" },

new() { AuthorName = "Halide Edip Şafak"}

}

};

await context.AddAsync(book);

await context.SaveChangesAsync();

class Book

{

public Book()

{

Authors = new HashSet<Author>();

}

public int Id { get; set; }

public string BookName { get; set; }

public ICollection<Author> Authors { get; set; }

}

class Author

{

public Author()

{

Books = new HashSet<Book>();

}

public int Id { get; set; }

public string AuthorName { get; set; }

public ICollection<Book> Books { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Book> Books { get; set; }

public DbSet<Author> Authors { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=KitapDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

}

\*/

**//2.Yöntem: n t n ilişkisi eğer ki fluent api ile tasarlanmışsa kulllanılan bir yöntemdir.**

/\*

Author author = new()

{

AuthorName = "Fatih Terimson",

Books = new HashSet<AuthorBook>()

{

new() {BookId = 1}, //hem varolan verilere yeni verileri ilişkilendirmek, (cross table entity'si üzerinden de işlem yapılabilir)

new() {Book = new() { BookName = "Parayla Şampuanlık" } } //hem de olmayan verileri ilişkisel bir şekilde eklemek istiyorsak.

}

};

await context.AddAsync(author);

await context.SaveChangesAsync();

class Book

{

public Book()

{

Authors = new HashSet<AuthorBook>();

}

public int Id { get; set; }

public string BookName { get; set; }

public ICollection<AuthorBook> Authors { get; set; }

}

class AuthorBook

{

public int BookId { get; set; }

public int AuthorId { get; set; }

public Book Book { get; set; }

public Author Author { get; set; }

}

class Author

{

public Author()

{

Books = new HashSet<AuthorBook>();

}

public int Id { get; set; }

public string AuthorName { get; set; }

public ICollection<AuthorBook> Books { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Book> Books { get; set; }

public DbSet<Author> Authors { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=KitapDB;User ID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<AuthorBook>()

.HasKey(ba => new { ba.AuthorId, ba.BookId });

modelBuilder.Entity<AuthorBook>()

.HasOne(ba => ba.Book)

.WithMany(b => b.Authors)

.HasForeignKey(ba => ba.BookId);

modelBuilder.Entity<AuthorBook>()

.HasOne(ba => ba.Author)

.WithMany(b => b.Books)

.HasForeignKey(ba =>ba.AuthorId);

}

}

\*/

**#endregion**

**DERS 10: İlişkisel Senaryolarda Veri Güncelleme**

using Microsoft.EntityFrameworkCore;

ApplicationDbContext context = new();

**#region One to One İlişkisel Senaryolarda Veri Güncelleme**

**#region Saving**

//Person person = new()

//{

// Name = "Gençay",

// Address = new()

// {

// PersonAddress = "Yenimahalle/ANKARA"

// }

//};

//Person person2 = new()

//{

// Name = "Hilmi"

//};

//await context.AddAsync(person);

//await context.AddAsync(person2);

//await context.SaveChangesAsync();

**#endregion**

**#region 1. Durum | Esas Tablodaki Veriye Bağımlı Veriyi Değiştirme**

/\*

//Öncelikle hedef Person'u ilgili Person'a karşılık gelen adres bilgisiyle birlikte elde etmem lazım.

//Person person = await context.Persons.FirstOrDefaultAsync(p => p.Id == 1); bu şekilde Id'si 1 olan Person'u getir dedik. Ama //adres bilgisi de lazım. Join yapıları lazım.

//Hedef Person'u elde ettik ve yanında adres bilgisini de getirmiş olduk.

Person? person = await context.Persons //? koyarak buradaki gelecek olan verinin nullable olabileğinin farkındayım, bilgin olsun //compiler bey, demiş oluyoruz.(uyarıyı gidermek için)

.Include(p => p.Address) //Include vermiş olduğumuz Navigation Property'e karşılık gelen tabloyla, arkada oluşturacağı select //sorgusunda bir join işlemi yapar.

.FirstOrDefaultAsync(p => p.Id == 1);

//EF Core'da Include'u kullanıyorsak vermiş olduğumuz NP her neyse ilgili verilere karşılık o NP'de değerler olacaktır. person. dediğimiz ilgili değerler gelecektir, ilgili Person'un adresinin nesnesi bağlanmış olacaktır -> person.Address

context.Addresses.Remove(person.Address); //eski adresi sildik

person.Address = new() //yerine yeni adres nesnesi ile yeni adres bilgisini ekledik.

{

PersonAddress = "Yeni adres bilgisi."

};

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 2. Durum | Bağımlı Verinin İlişkisel Olduğu Ana Veriyi Güncelleme**

//Address? address = await context.Addresses.FindAsync(1); //1 Id'sine sahip olan adresi elde ettik.

//address.Id = 2; //Adresin içindeki Id kolonu bir key'e karşılık geldiğinden dolayı silinemez, değiştirilemez.

//await context.SaveChangesAsync(); //hata.

//Öncelikle mevcut olan bağımlı-dependent veriyi sil. Sildikten sonra SaveChanges'ı çağır.

//Address? address = await context.Addresses.FindAsync(1); //adresi aldık.

//context.Addresses.Remove(address); //1 Id'sine sahip adresi sildik.

//await context.SaveChangesAsync();

//Ondan sonra tekrardan ilgili veriyi oluşturduktan sonra Principal veriyle bunu ilişkilendir. (Mevcut Person ile ilişkilendir.)

//Person? person = await context.Persons.FindAsync(2); //adresin bilgilerini yeniden oluştur, (adres InMemory'de duruyor, sadece tablodan silindi.)

//address.Person = person; //ve bunu person ile ilişkilendir.

//Adresimize yeni bir Person ekleyerek de ekleme operasyonumuzu gerçekleştirebiliriz.

/\*

Address? address = await context.Addresses.FindAsync(2);

context.Addresses.Remove(address);

await context.SaveChangesAsync();

address.Person = new()

{

Name = "Faruk"

};

await context.Addresses.AddAsync(address);

await context.SaveChangesAsync();

\*/

#endregion

#endregion

#region One to Many İlişkisel Senaryolarda Veri Güncelleme

#region Saving

/\*

Blog blog = new()

{

Name = "noktadotkom.com Blog",

Posts = new List<Post>

{

new(){ Title = "1. Post" },

new(){ Title = "2. Post" },

new(){ Title = "3. Post" },

}

};

await context.Blogs.AddAsync(blog);

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 1. Durum | Esas Tablodaki Veriye Bağımlı Verileri Değiştirme**

//Önce Blog'u sonra Blog'la ilişkisel durumda olan Post'ları elde edeceğiz. (join işlemi)

/\*

Blog? blog = await context.Blogs

.Include(b => b.Posts) //Post'ları da sorguya ekle.

.FirstOrDefaultAsync(b => b.Id == 1); //1 Id'sine sahip Blog'u elde et.

Post? silinecekPost = blog.Posts.FirstOrDefault(p => p.Id == 2); //2 Id'sine sahip silmek istediğimiz Post'u elde et.

blog.Posts.Remove(silinecekPost); //ilgili Post'u sil.

blog.Posts.Add(new() { Title = "4. Post" }); //4 ve 5 Post Title'larına sahip olan yeni Postları Blog'la ilişkilendir.

blog.Posts.Add(new() { Title = "5. Post" });

await context.SaveChangesAsync();

\*/**#endregion**

**#region 2. Durum | Bağımlı verilerin ilişkisel olduğu ana veriyi güncelleme**

//4.Post'u BlogId'si 2 olan Blog ile ilişkilendirelim. Yanlışlıkla 1'e gitmiş varsayıp 2'ye çevirelim.

/\*

Post? post = await context.Posts.FindAsync(4); //hangi Post'un foreign key'ini güncelleyeceksek ona geliyoruz. 4 Id'sine sahip Post'u elde ettik.

post.Blog = new() //Blog 2 olmadığından dolayı yenisini oluşturalım.

{

Name = "2. Blog"

};

//salt bir güncelleme olacağından dolayı direkt olarak SaveChanges'i çağırabiliriz. 2. Blog'u oluşturup ardından hangi Post'taysak //onu da 2 ile güncelle.

await context.SaveChangesAsync();

\*/

//Peki 5.Post'u da 2'ye vermek istersek? Yukarıdaki kodu çalıştırdığımız için artık 2. Blog mevcut.

/\*

Post? post = await context.Posts.FindAsync(5); //5.Post'u elde et.

Blog? blog = await context.Blogs.FindAsync(2); //2.Blog'u elde et.

post.Blog = blog; //2.Blog'u 5.Post'a ver.

await context.SaveChangesAsync();

\*/

**#endregion**

**#endregion**

**#region Many to Many İlişkisel Senaryolarda Veri Güncelleme**

**#region Saving**

/\*

Book book1 = new() { BookName = "1. Kitap" };

Book book2 = new() { BookName = "2. Kitap" };

Book book3 = new() { BookName = "3. Kitap" };

Author author1 = new() { AuthorName = "1. Yazar" };

Author author2 = new() { AuthorName = "2. Yazar" };

Author author3 = new() { AuthorName = "3. Yazar" };

book1.Authors.Add(author1); //1.Kitapla 1 ve 2. Yazarları

book1.Authors.Add(author2);

book2.Authors.Add(author1); //2. Kitapla 1-2-3. Yazarları

book2.Authors.Add(author2);

book2.Authors.Add(author3);

book3.Authors.Add(author3); //3. Kitapla 3. Yazarı ilişkilendirip,

await context.AddAsync(book1); //Eklemeleri yaptık.

await context.AddAsync(book2);

await context.AddAsync(book3);

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 1. Örnek: 1. Kitaba 3. Yazarı da eklemek-ilişkilendirmek istersek;**

/\*

Book? book = await context.Books.FindAsync(1); //1. kitabı elde et.

Author? author = await context.Authors.FindAsync(3); //3. yazarı elde et.

book.Authors.Add(author); //1. book'un Authors'una 3. Author'u ekle.

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 2. Örnek :3 Id'sine sahip yazarın sadece 1 Id'sine sahip kitapla ilişkisi olsun. Diğer ilişkilerini koparalım.**

/\*

Author? author = await context.Authors

.Include(a => a.Books) //Yazarın ilişkisi olduğu kitapları elde et.

.FirstOrDefaultAsync(a => a.Id == 3); //3 Id'sine sahip olan yazarı elde et.

foreach (var book in author.Books) //foreach ile yazarın sahip olduğu bütün Books'lara girelim. Book'ları elde edip silelim. (tek tek manuel de yapılabilir bu işlem)

{

if (book.Id != 1) //Id'si 1 değilse book'un

author.Books.Remove(book); //Book'ları sil.

}

await context.SaveChangesAsync();

\*/

**#endregion**

**#region 3. Örnek: 2 Id'sine sahip kitabın, 1 Id'sine sahip yazarla ilişkisini keselim 3 Id'sine sahip yazarla ilişkisini ekleyelim. Ekstradan 4. Yazarı da ekleyelim.**

Book? book = await context.Books //kitaptan yola çıktığımız için Book ile başlayalım.

.Include(b => b.Authors) //İlgili book'a karşılık gelen tüm yazarları elde et.

.FirstOrDefaultAsync(b => b.Id == 2); //2 Id'sine sahip Book'u elde et.

Author silinecekYazar = book.Authors.FirstOrDefault(a => a.Id == 1); //book. üzerinden 1 Id'sine sahip olan Author'u elde et.

book.Authors.Remove(silinecekYazar); //ve cross table'dan kaydını-ilişkisini sil-kopar.

Author eklenecekYazar = await context.Authors.FindAsync(3); //3 Id'sine sahip Author'u elde et.

book.Authors.Add(eklenecekYazar); //book. üzerinden yeni ilişkiyi ekle.

//yeni 4.yazarı ekle ve 2 Id'sine sahhip kitapla ilişkilendir.

book.Authors.Add(new() { AuthorName = "4. Yazar" });

await context.SaveChangesAsync();

**#endregion**

**#endregion**

class Person

{

public int Id { get; set; }

public string Name { get; set; }

public Address Address { get; set; }

}

class Address

{

public int Id { get; set; }

public string PersonAddress { get; set; }

public Person Person { get; set; }

}

class Blog

{

public Blog()

{

Posts = new HashSet<Post>();

}

public int Id { get; set; }

public string Name { get; set; }

public ICollection<Post> Posts { get; set; }

}

class Post

{

public int Id { get; set; }

public int BlogId { get; set; }

public string Title { get; set; }

public Blog Blog { get; set; }

}

class Book

{

public Book()

{

Authors = new HashSet<Author>();

}

public int Id { get; set; }

public string BookName { get; set; }

public ICollection<Author> Authors { get; set; }

}

class Author

{

public Author()

{

Books = new HashSet<Book>();

}

public int Id { get; set; }

public string AuthorName { get; set; }

public ICollection<Book> Books { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Person> Persons { get; set; }

public DbSet<Address> Addresses { get; set; }

public DbSet<Post> Posts { get; set; }

public DbSet<Blog> Blogs { get; set; }

public DbSet<Book> Books { get; set; }

public DbSet<Author> Authors { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ApplicationDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

.HasForeignKey<Address>(a => a.Id);

}

}

**DERS 11: İlişkisel Senaryolarda Veri Silme**

using Microsoft.EntityFrameworkCore;

using System.Runtime.Intrinsics.X86;

ApplicationDbContext context = new();

**#region One to One İlişkisel Senaryolarda Veri Silme**

/\*

Person? person = await context.Persons //Persons'a git.

.Include(p => p.Address) //Address ile join'leyip çek.

.FirstOrDefaultAsync(p => p.Id == 1); //oradan da 1 Id'sine sahip Person hangisiyse elde et.

if (person != null) //person null değilse aşağıdaki operasyonu gerçekleştir. Olmayan bir şeyi silemeyiz sonuçta.

context.Addresses.Remove(person.Address); //İlgili adresi sil.

//1 Id'sine sahip Person'un ilişkili adresini sildik.

await context.SaveChangesAsync();

\*/

**#endregion**

**#region One to Many İlişkisel Senaryolarda Veri Silme**

//2 Id'sine sahip Post silinsin. 1 ve 3 kalsın.

/\*

Blog? blog = await context.Blogs //hedef Blog hangisiyse onu elde etmemiz lazım.

.Include(b => b.Posts) //İlgili Blog'a karşılık gelen bütün Post'ları sorguya ekle.

.FirstOrDefaultAsync(b => b.Id == 1); //1 Id'sine sahip Blog.

Post? post = blog.Posts.FirstOrDefault(p => p.Id == 2); //2 Id'sine sahip Post'u elde et.

context.Posts.Remove(post); //Ve sil.

await context.SaveChangesAsync();

\*/

**#endregion**

**#region Many to Many İlişkisel Senaryolarda Veri Silme**

//Buradaki işlemler cross table üzerinden yapılır.

//1 kitabına karşılık gelen 2 Yazarını silmeye çalışalım. Burada Yazar komple silinmez. Cross Table'daki ilişkisi silinir.

/\*

Book? book = await context.Books

.Include(b => b.Authors)

.FirstOrDefaultAsync(b => b.Id == 1);

Author? author = book.Authors.FirstOrDefault(a => a.Id == 2);

//context.Authors.Remove(author); //Yazarı silmeye kalkar!

book.Authors.Remove(author); //bağı koparmış oluruz. 1 Id'sine sahip kitabın artık 2 Id'li yazarla bir ilişkisi yok.

await context.SaveChangesAsync();

\*/

/\*

Book? book = await context.Books

.Include(b => b.Authors)

.FirstOrDefaultAsync(b => b.Id == 2); //2 Id'sine sahip Book'u getir.

Author? author = book.Authors.FirstOrDefault(a => a.Id == 2); //Id'sine sahip olan Kitaptan 2 Id'sine sahip olan yazarı elde et.

context.Authors.Remove(author); //Yazarı silmeye kalkar! Hem ilişki hem de yazarı sildi. Veri kaybı söz konusu.

await context.SaveChangesAsync();

\*/

**#endregion**

//Şu ana kadar Principle Table'daki herhangi bir verinin Dependent Table'daki ilişkisel verileri arasından birilerini silmeye //çalışırken nasıl davranış sergileyeceğimizi gördük.

**//Principle Table'daki bir veriyi silmeye çalışırsak;**

**#region Cascade Delete Yapılanması**

//Aşağıdaki davranış modelleri Fluent API ile konfigüre edilebilmektedir.

#region Cascade

//Esas tablodan silinen veriyle karşı/bağımlı tabloda bulunan ilişkili verilerin silinmesini sağlar.

/\*

Blog? blog = await context.Blogs.FindAsync(1);

context.Blogs.Remove(blog); //Hem Blog hem de ilişkili post'lar silindi.

await context.SaveChangesAsync();

\*/

**#endregion**

**#region SetNull**

//Esas tablodan silinen veriyle karşı/bağımlı tabloda bulunan ilişkili verilere null değerin atanmasını sağlar.

/\*

Blog? blog = await context.Blogs.FindAsync(1);

context.Blogs.Remove(blog); //Hem Blog hem de ilişkili post'lar silindi.

await context.SaveChangesAsync();

\*/

//Aşağıda detaylı not:

//One to One senaryolarda eğer ki Foreign key ve Primary key kolonları aynı ise o zaman SetNull davranışını KULLANAMAYIZ!

**#endregion**

**#region Restrict**

//Esas tablodan herhangi bir veri silinmeye çalışıldığında o veriye karşılık dependent table'da ilişkisel veri/veriler varsa eğer bu sefer bu silme işlemini engellenmesini sağlar.

/\*

Blog? blog = await context.Blogs.FindAsync(1);

context.Blogs.Remove(blog);

await context.SaveChangesAsync();

//hata alırız. çünkü ilişkisel verilerimiz var. Restrict koyduk. Silemeyiz.

\*/

**#endregion**

**#endregion**

**#region Saving Data**

//Person person = new()

//{

// Name = "Berk",

// Address = new()

// {

// PersonAddress = "Yenimahalle/ANKARA"

// }

//};

//Person person2 = new()

//{

// Name = "Hilmi"

//};

//await context.AddAsync(person);

//await context.AddAsync(person2);

//Blog blog = new()

//{

// Name = "yildizkenter.com Blog",

// Posts = new List<Post>

// {

// new(){ Title = "1. Post" },

// new(){ Title = "2. Post" },

// new(){ Title = "3. Post" },

// }

//};

//await context.Blogs.AddAsync(blog);

//Book book1 = new() { BookName = "1. Kitap" };

//Book book2 = new() { BookName = "2. Kitap" };

//Book book3 = new() { BookName = "3. Kitap" };

//Author author1 = new() { AuthorName = "1. Yazar" };

//Author author2 = new() { AuthorName = "2. Yazar" };

//Author author3 = new() { AuthorName = "3. Yazar" };

//book1.Authors.Add(author1);

//book1.Authors.Add(author2);

//book2.Authors.Add(author1);

//book2.Authors.Add(author2);

//book2.Authors.Add(author3);

//book3.Authors.Add(author3);

//await context.AddAsync(book1);

//await context.AddAsync(book2);

//await context.AddAsync(book3);

//await context.SaveChangesAsync();

**#endregion**

class Person

{

public int Id { get; set; }

public string Name { get; set; }

public Address Address { get; set; }

}

class Address

{

public int? Id { get; set; }

public string PersonAddress { get; set; }

public Person Person { get; set; }

}

class Blog

{

public Blog()

{

Posts = new HashSet<Post>();

}

public int Id { get; set; }

public string Name { get; set; }

public ICollection<Post> Posts { get; set; }

}

class Post

{

public int Id { get; set; }

public int? BlogId { get; set; } //SetNull ile çalışabilmek için Nullable yapmamız lazım.

public string Title { get; set; }

public Blog Blog { get; set; }

}

class Book

{

public Book()

{

Authors = new HashSet<Author>();

}

public int Id { get; set; }

public string BookName { get; set; }

public ICollection<Author> Authors { get; set; }

}

class Author

{

public Author()

{

Books = new HashSet<Book>();

}

public int Id { get; set; }

public string AuthorName { get; set; }

public ICollection<Book> Books { get; set; }

}

class ApplicationDbContext : DbContext

{

public DbSet<Person> Persons { get; set; }

public DbSet<Address> Addresses { get; set; }

public DbSet<Post> Posts { get; set; }

public DbSet<Blog> Blogs { get; set; }

public DbSet<Book> Books { get; set; }

public DbSet<Author> Authors { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=PC\SQLEXPRESS;Database=ApplicationDB;UserID=sa;Password=1;TrustServerCertificate=True;Trusted\_Connection=true");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

/\*Birebire ve bireçok durumlarında Cascade.

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

.HasForeignKey<Address>(a => a.Id)

.OnDelete(DeleteBehavior.Cascade); //Eğer ki Person'dan herhangi bir veri silinirse buna karşılık Address tablosunda ilişkisel data varsa onu da sil.

//EF Core default olarak ilişkisel tablolar arasındaki silme davranışı Cascade olarak ayarlar.

modelBuilder.Entity<Post>()

.HasOne(p => p.Blog)

.WithMany(b => b.Posts)

.OnDelete(DeleteBehavior.Cascade); //Eğer ki Blog'dan herhangi bir veri silinirse buna karşılık Post tablosunda ilişkisel data varsa onu da sil.

\*/

/\*

//Birebir ve bireçok durumlarında SetNull

//SetNull olarak değiştirdiğimizde yeni migration'u oluşturmayı unutma.

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

.HasForeignKey<Address>(a => a.Id);

//.OnDelete(DeleteBehavior.SetNull); olmaz, hata verir. Birebir yapılarda SetNull vereceksek, Foreign Key kolonu ayrı bir property tarafından temsil edilmeli.

//Cannot create the foreign key "FK\_Addresses\_Persons\_Id" with the SET NULL referential action, because one or more //referencing columns are not nullable.

//Could not create constraint or index. See previous errors. Update-database dediğimizde bu hatayı alırız. Çünkü Id'yi hem //Primary hem Foreign Key olarak kullandık.

//İşte bu yüzden birebir durumlarında SetNull Kullanamayız.

//bireçok durumlarında kullanılabilir.

//Blog silindiği taktirde Post'un foreign key kolununa null basılması lazım bu durumda class Post'a gidip int? BlogId

modelBuilder.Entity<Post>()

.HasOne(p => p.Blog)

.WithMany(b => b.Posts)

.OnDelete(DeleteBehavior.SetNull)

.IsRequired(false);//ilgili foreign key kolonu illa ki required olmak zorunda değil.

\*/

//Birebir ve bireçok'ta Restrict.

/\*

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

.HasForeignKey<Address>(a => a.Id);

modelBuilder.Entity<Post>()

.HasOne(p => p.Blog)

.WithMany(b => b.Posts)

.OnDelete(DeleteBehavior.Restrict)

.IsRequired(false);

\*/

//Çokaçok ilişkide her daim silme davranışı cascade üzerinde kuruludur. Yukarıdaki davranuş modelleri EF Core'da çokaçok ilişkilerde izin verilen silme yöntemleri değildir.

}

}