**ORM Nedir?**

ORM (Object Relational Mapping) veritabanı tablolarını ve ilişkilerini nesnelerle eşleştirerek ve bu nesneler üzerinden iletişim kurmayı esas alan yaklaşımdır.

Yani veritabanını doğrudan yazılım projesi içerisinde işlemek yerine oluşturulan nesneler üzerinden işlemeyi amaçlar.

**EFCore Nedir?**

ORM yaklaşımını benimsemiş bir araçtır. Kod içerisinde OOP ile SQL sorguları oluşturmamızı sağlar. Tekrarlı SQL sorgucuklarından bizleri kurtarır. Code First ve Database First yaklaşımları eşliğinde veritabanı ile yazılım arasındaki koordinasyonu sağlar.

**EFCore Nasıl Yüklenir?**

CLI

* cmd 🡺 **dotnet tool install --global dotnet-ef**
* Tool yüklendikten sonra herhangi bir projede kullanabilmek için **Microsoft.EntityFrameworkCore.Design** paketinin projede yüklü olması gerekmektedir.

PMC

* Visual Studio, PMC üzerinden talimatlar verebilmek için **Microsoft.EntityFrameworkCore.Tools** paketinin ilgili projede yüklü olması gerekmektedir.

**DBFirst ve CodeFirst**

EFCore, veritabanı çalışmaları için, veritabanının önceden olup olmaması durumlarına göre farklı yaklaşımlar sunar. Dolayısıyla hedef veritabanının kod kısmına aktarılması gerekmektedir. Lakin hedef veritabanın önceden oluşturulmuş mu yoksa yeni mi oluşturulacağı önem arz etmektedir. Sonuç olarak EFCore’un bize sunmuş olduğu yaklaşımlar veritabanının var olup olmaması durumlarına göre kod kısmında modellenme süreçlerinin hangi davranışla gerçekleştirileceğini belirleyecek olan tekniği bizlere sunmaktadır.

Veritabanı var olan bir projedeki yaklaşım büyük ihtimalle DBFirst, veritabanı inşa edilmemiş bir projede DBFirst ya da CodeFirst yaklaşımları tercih edilir.

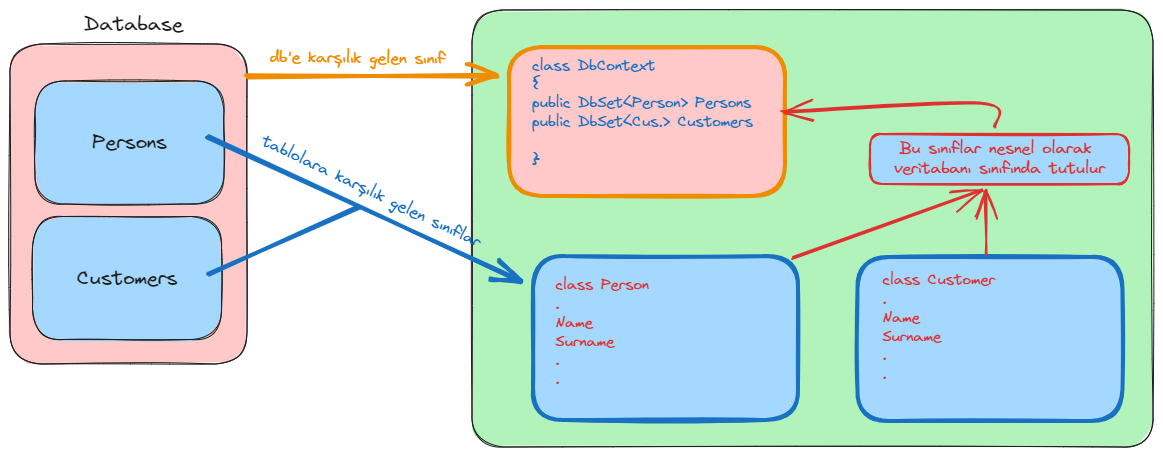
Yani bir projede hangi yaklaşımın benimseneceği birkaç unsura bağlı olarak değerlendirilir.

Hangi yaklaşımım benimsenmesine karar vermek için bu yaklaşımların tam teferruatlı bilmek ve değerlendirmek gerekir.

**DBFirst Yaklaşımı**

EFCore ile işlem yapılacak veritabanı önceden var ise kesin olarak DBFirst yaklaşımını benimsenmemiz gerektiğini söyleyebiliriz.

DBFirst var olan veritabanını analiz ederek kod kısmında modelleyen bir yaklaşımdır.

* AVANTAJLARI

1. Hazır veritabanları hızlıca modellenebilir
2. Veritabanında süreçte olan değişikliklere göre hızlıca koda aktarmamızı sağlar
3. EFCore ile desteklenen tüm veritabanlarında kullanılabilir.

* DEZAVANTAJLARI

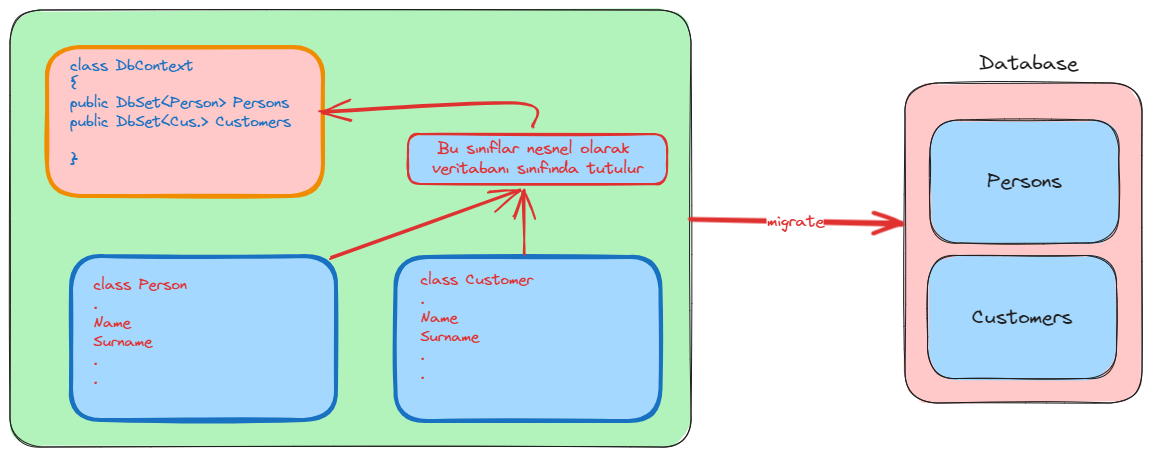
1. Kod veritabanına göre şekilleneceği için yönetim veritabanı tarafından sağlanır yani veritabanı bilgisi gerektirir.
2. Değişiklikler veritabanında olacağı için geliştirici tarafından sürekli kontrol davranışı sergilenmelidir.

* Hangi Durumlarda Tercih Edilmelidir?
* Önceden oluşturulmuş, veritabanı var olan uygulamalarda.
* Veritabanı yönetimine, geliştirme süreçlerine ve tasarımına dair herhangi bir kararın geliştiriciler tarafından verilmediği durumlarda tercih edilmelidir.

DBFirst, CodeFirst yaklaşımının tam tersidir veya CodeFirst,DBFirst yaklşaımının tam tersidir diyebiliriz.

**CodeFirst Yaklaşımı**

EFCore ile çalışma yapılacak olan veritabanı yok ise yani henüz oluşturulmamış ise bu veritabanını geliştirme aşamasında modelleyerek veritabanını sunucuda oluşturan (migration) yaklaşımdır.

Bu yaklaşımda veritabanı önce kodla tasarlanır, sonra db sunucusuna gönderilerek veritabanı oluşturulur.

* AVANTAJLARI

1. Kod üzerinde veritabanı modellememizi sağlar.
2. Veritabanına dokunmadan kod üzerinden veritabanındaki gerekli düzenlemelerin yapılmasını sağlar.
3. EFCore ile desteklenen tüm veritabanlarında kullanılabilir.
4. Koddaki ihtiyaca dönük veritabanı şekilleneceği için herhangi bir veritabanı bilgisine gerek duyulmamaktadır.
5. Değişiklikler kod kısmından yapılacağı için herhangi bir kontrol ihtiyacı söz konusu değildir.
6. Veritabanı modeli kod üzerinde yapıldığı için istenilen sunucuda anında ilgili modelin değerleri elde edilebilir.

* DEZAVANTAJLARI
  1. Üretilecek veritabanının tasarımı ve stratejisi geliştirici sorumluluğundadır.
* Hangi Durumlarda Tercih Edilmelidir
  1. Veritabanı bilgisine ihtiyaç duyulamayan
  2. Veritabanı tasarımının geliştirici tarafından üstlenilebileceği
  3. Veritabanı yönetiminin kod üzerinden sağlanacağı durumlarda tercih edilebilir.

**BAŞLIK**

* Bir ORM aracının veritabanını temsil edebilmesi için o veritabanındaki tabloların, tablolar içerisindeki kolon ve nesnelerin modellenmesi gerekmektedir.
* Bu modelleme class’lar üzerinden gerçekleştirilir.

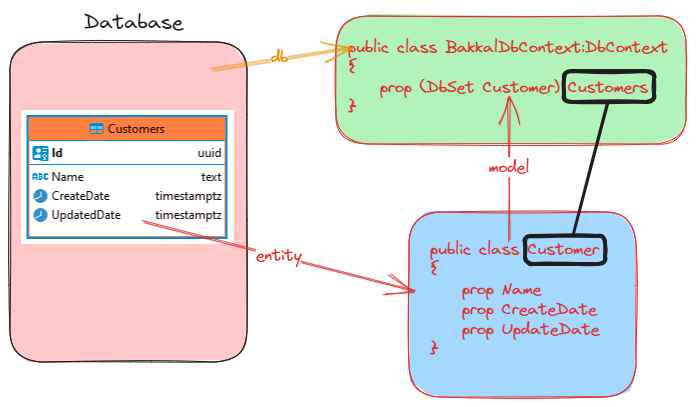
VERİTABANI CLASSI: DbContext

* + EFCore da veritabanını temsil eden sınıf DBContext olarak nitelendirilmektedir.
  + Bir sınıfın veritabanına karşılık gelen sınıf olabilmesi için Microsoft.EntityFrameworkCore namespace’i altındaki DbContext sınıfından türemesi gerekmektedir.
  + Bakkal adında bir veritabanımız olsun ismi şu şekilde olacaktır: BakkalDbContext.

DbContext aşağıdaki faaliyetlerden sorumludur;

1. Konfigurasyon: Veritabanı bağlantısı, model yapılanmaları ve veritabanı nesnesi ile tablo nesneleri arasındaki ilişkileri sağlar
2. Sorgulama: Kod tarafında gerçekleştirilen sorgulama adımlarını SQL sorgusuna dönüştürür ve veritabanına gönderir.
3. Change Tracking: Sorgulama neticesinde veriler üzerindeki değişiklikleri takip eder
4. Veri Kalıcılığı: Verilerin kaydedilmesi, silinmesi veya güncellenmesi gibi operasyonları gerçekleştirir.
5. Caching

TABLO CLASSI: Entity

* + EFCore da tabloları temsil eden sınıflar entity olarak nitelendirilir.
  + Herhangi bir olguyu/nesneyi modelleyen sınıfa entity denir.
  + Tüm entity sınıfları DbContext sınıfı içerisine, DbSet olarak eklenmelidir. Böylece veritabanı sınıfı ile entity sınıfı arasında bir ilişki kurulacak ve EFCore, veritabanındaki tabloları hangi sınıflarla temsil edildiği bu ilişki üzerinden anlayacaktır.
  + DbSet<T> şeklinde DbContext içerisinde tanımlanır. Buradaki T tabloyu temsil edecek olan modeli içermelidir.
  + Veritabanında tabloyu temsil edecek olan property çoğul isimde olurken tablo içerisindeki model tekil isimde olacaktır. Yani Customers adlı bir tabloyu Customer adlı bir modelle temsil edeceğiz diyebiliriz. Dolayısıyla Customers içerisinde birden fazla Customer barındırabilir.
  + EFCore’da bir tabloyu temsil eden sınıf içerisindeki propertyler o tabloya ait kolonları temsil eder.
  + Veritabanındaki veriler entitylerin instancelarına karşılık gelmektedir.

**DbFirst Pratik**

İlgili veritabanının kod kısmında oluşturulması gerekmektedir. Bu işlemi PMC veya Dotnet CLI aracılığı ile iki farklı şekilde yürütebiliriz.

Connection strings: [https://www.connectionstrings.com]

PMC ile 🡺

Scaffold-DbContext ‘ConnectionString’ Microsoft.EntityFrameworkCore –[Provider]

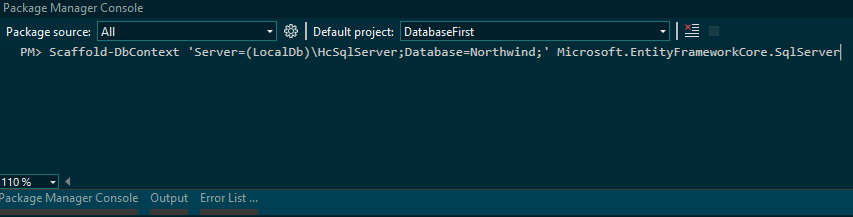
ConnectionString:Server=myServerAddress;Database=myDataBase;User Id= myUsername; Password=myPassword;

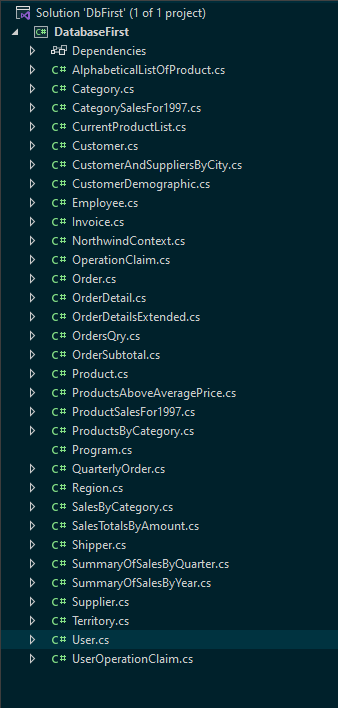
Yüklenmesi gereken kütüphaneler:

* Microsoft.EntityFrameworkCore.Tools
* Database Provider

[İlgili databaselerin provideslerına alttaki linkenten erişebilirsiniz

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





İlgili talimatları PMCden verdiğimizde veritabanımızın sınıfları ve Contextimiz oluşturulmuş olacaktır.

Dotnet CLI ile 🡺

dotnet ef dbcontext scaffold ‘ConnectionString’ [Provider]

Yüklenmesi gereken kütüphaneler:

* Microsoft.EntityFrameworkCore.Design
* Database Provider

cmd üzerinden:

dotnet ef dbcontext scaffold Server=(LocalDb)\HcSqlServer; Database= Northwind; Microsoft.EntityFrameworkCore.SqlServer

Talimatı verildiğinde veritabanımızın sınıfları ve Contextimiz oluşturulmuş olacaktır.

Varsayılan olarak databasedaki bütün tablolar modellenir. Ancak biz istenilen tablolaların modellenmesini istiyorsak aşağıdaki talimatları vermemiz gerekmekte:

Scaffold-DbContext ‘CS’[Provider] -Tables TableName1, TableName2…

dotnet ef dbcontext scaffold ‘CS’[Provider] --table TableName,--table TableName2..

TEKRAR TEKRAR YAZMAMAK ADINA “Scaffold-DbContext ‘CS’[Provider]” ve “dotnet ef dbcontext scaffold ‘CS’[Provider]” talimatlarını “Directive” olarak kısalttım

Eğerki oluşturulan context nesnesini adını değiştirmek istiyorsak;

“Directive” -Context ContextName (PMC)

“Directive” --context ContextName (CLI)

Path ve Namespace Belirtme

Entityler ve DbContext sınıfı varsayılan olarak projenin varsayılan namespaceni kullanır eğer ki bunlara müdahale etmek istiyorsanız aşağıdaki gibi talimat verebiliriz:

PATH:

“Directive” –ContextDir [Context Dosya Adı] –OutputDir [Model Dosya Adı] (PMC)

“Directive” –-[^^] Data –-[^^] Models (CLI)

[] içine yazılmamalı

NAMESPACE:

“Directive” -Namespace NameSpaceAdı -ContextNamespace NameSpaceAdı

“Directive” --namespace NameSpaceAdı --context-namespace NameSpaceAdı

Model Güncelleme

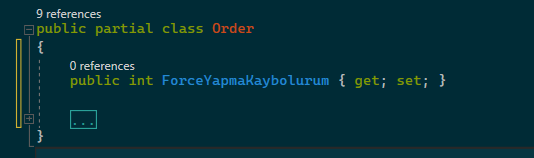
Eğer veritabanında kod yazma aşamasında bir güncelleme olurda veritabanı güncellenirse modellerimizi güncellememiz gerekebilir

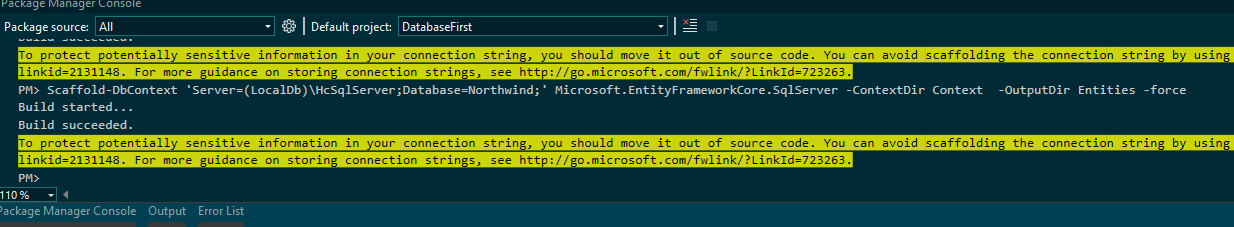
Haliyle “Directive” kullanıldığında hata alınacaktır.

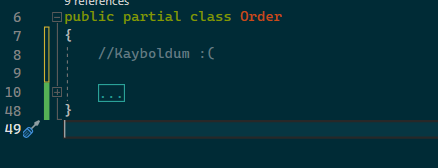
Bunun için “Directive” -Force ile güncelleyebiliriz

Modellerin Özelleştirilmesi

Modeller özelleştirildiğinde örneğin Order classına bir değer daha eklediğimizde -force komutu kullanıldığında ilgili değişiklik ezilecek yani silinecektir. Bu sorunu yaşamamak adına ilgili değişiklikleri yada özelleştirmeleri partial classlar altında yapmak en doğrusudur. Aksi halde özelleştirmeler ezilecektir.





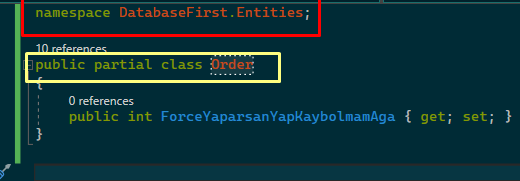


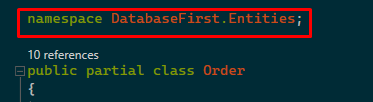
Haliyle buradaki propertyimiz merkezi bir yapılanmada olabilir ve bizi mağduriyete sürüklememesi açısından;

HATIRLATMA AYNI İSİMDE AYNI NAMESPACE ALTINDA BİRDEN FAZLA PARTIAL CLASS TUTULABİLİR



Entitylerin altında aynı namespace altında aynı isimde classımızı oluşturup propertymizi tanımlıyoruz





Haliyle force yapıldığında ilgili property ezilmeyecektir.

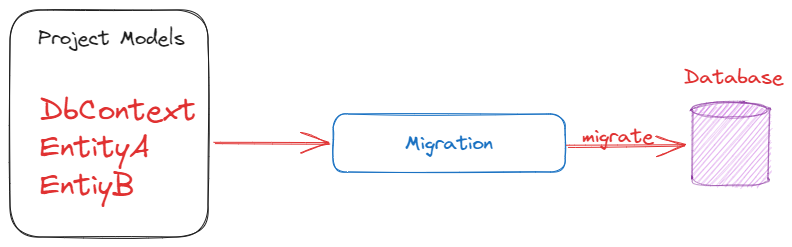
**CodeFirst Pratik**

**Migration & Migrate:**

Modellediğimiz classların veritabanı tablolarına dönüştürülme aşamasında bir ara birime ihtiyacımız vardır buradaki arabirim Migration partial classıdır.

Yani Migration partial classı bizim model classlarımızı veritabanının anlayacağı hale dönüştüren yapılanmadır.

İlgili entityler modellendikten sonra modelle alakalı migration oluşturulmalıdır. Ardından oluşturulan migration veritabanına gönderilmelidir. Bu işleme migrate etme denir. Bu işlem sonucunda modellediğimiz entityler veritabanında da modellenecektir.



Migration Oluşturmak;

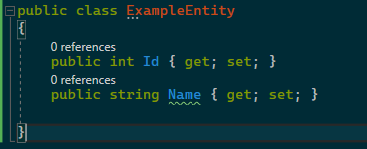
Migration oluştururken PMC veya CLI kullanabiliriz.

Yüklenmesi gereken paketler:

* Microsoft.EntityFramework.Core

Entity Oluşturmak;

Entityler class olarak içerisinde her bir kolonu temsil edecek olan propertyleri barındırır. Basit bir entity Modeli aşağıdaki gibidir.

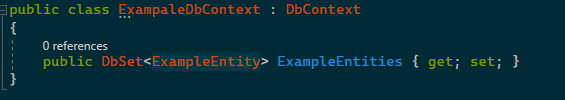


DbContext Oluşturmak;

Bir classın DbContext olması için DbContext sınıfından kalıtım alması gerekmektedir.

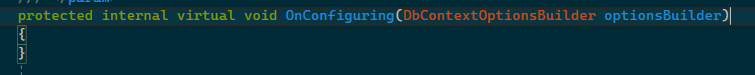
Oluşturulacak olan tabloların DbSet türünde bir nesne olacağını bildirmemiz gerekmektedir. DbSet abstract bir generic classdır haliyle oluşturalacak olan birimin türünü almaktadır.

Basit bir DbContext modeli aşağıdaki gibidir:



ExampleDbContext sınıfının kalıtım aldığı DbContext sınıfı Microsoft.EntityFramework.Core paketinden gelmektedir.

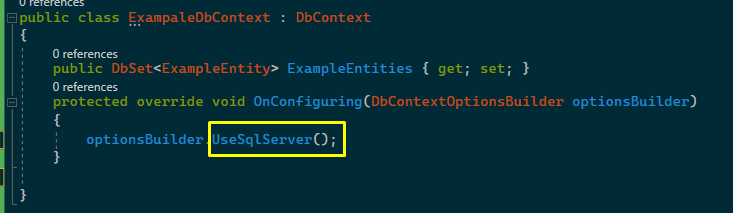
DbContext sınıfı içerisinde virtual bir method olan OnConfiguring methodu vardır.



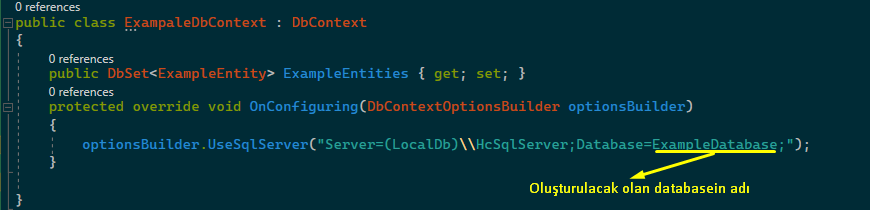
Bu methodu DbContext nesnemiz olan ExapmleDbContextde override ederek veritabanımızla alakalı konfigürasyonları yapmamızı sağlayan methoddur.

Bu methodu kullanarak oluşturduğumuz contextin hangi veritabanına migrate edileceğini bildiririz.

OnConfiguring methodu parametre olarak DbContextOptionsBuilder nesnesinden bir instance almaktadır. OptionsBuilder instance ile databaseimizin providerını verebiliriz.



İlgili providerın paketinin yüklü olması gerektiğini unutmayınız.



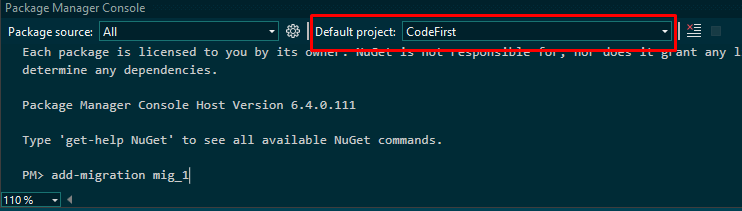
İlgili methoda bağlantı cümleciğini (ConnectionString) verdikten sonra migration oluşturabiliriz.

Migration Oluştururken:

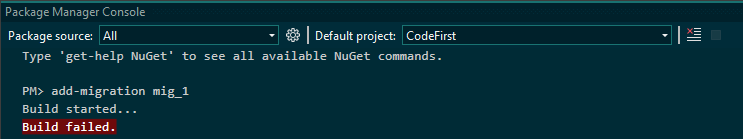
PMC ve CLI kullanabiliriz her iki yöntem içinde Microsoft.EntityFrameworkCore.Tools paketi projede yüklü olmalıdır.

PMC: add-migration [Migration Adı] [] kullanma

CLI: add-migrations [Migration Adı]



İlgili migration oluşturulurken Proje dizini olarak DbContext sınıfının bulunduğu dizin seçilmelidir.



CodeFirst dosyasına sağ tıklayıp 🡺 Properties 🡺 OutputType 🡺 Class Library olarak seçiniz

Could not load assembly ' PROJECT\_NAME'. Ensure it is referenced by the startup project ‘PROJECT\_NAME’.

Bu şekilde bir hata alıyorsanız ilgili projenize sağ tıklayarak Set as Starup Project olarak seçiniz

Sonuç olarak Migrationumuz oluştuğunda, Migration classından kalıtım almış bir sınıf oluşacaktır. Sınıfımız içerisinde iki adet fonskiyon göreceğiz bu fonksiyonlar Up ve Down fonksiyonlarıdır.

Up Fonksiyonu: Migrate sürecinde veritabanına gönderilecek olan bütün yapılanmalar bulunur.

Down Fonksiyonu: Veritabanında yapılan değişikliklerden hangilerinin geri alınacağını bildiren yapılanmaları barındırır.

Up inkılapçı iken Down irticacıdır 😀.



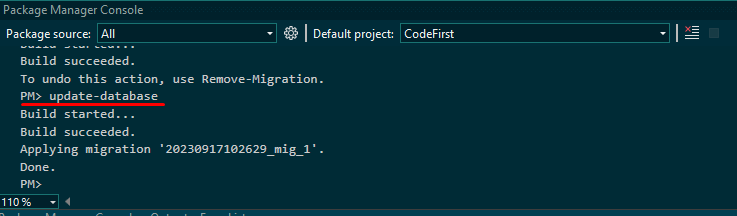
remove-migration ile en son oluşturulan migration silinir.

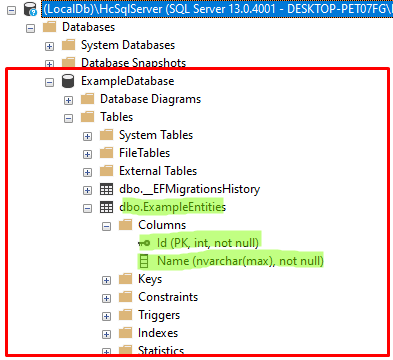
get-migration ile tüm migrationlar listelenir.

Migrationları Migrate etme;

updata-database talimatı ile migrationumuzu veritabanına migrate edilir

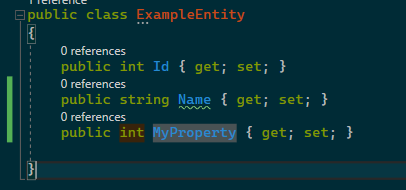
Sonuç olarak veritabanımız oluşturulacaktır.



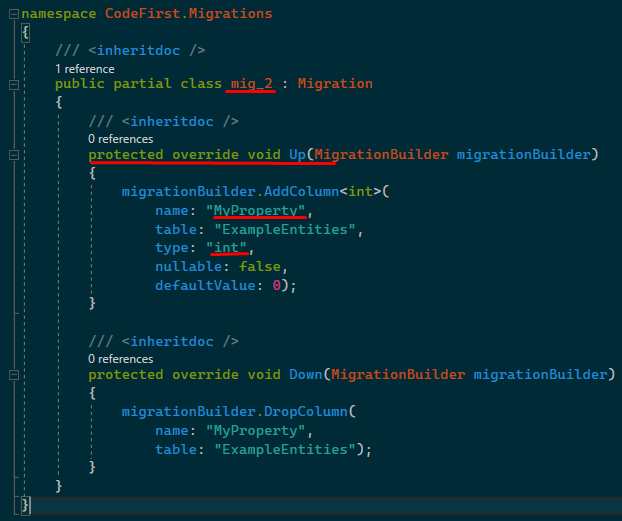


EFMigrationsHistory tablosu içersinde en son migrate migrationlara kadar migrate edilmiş migrationslar bulunur

Migration Geri Alma:



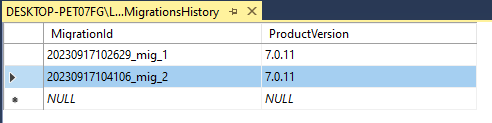
İlgili entitye bir property daha ekleyip yeni bir migrations oluşturalım.



update-database talimatını verdiğimizde ilgili kolon veritabanına eklenecektir.



MigrationsHistory Tablosuna baktığımızda en son migrate edilmiş migration en altta olacaktır.

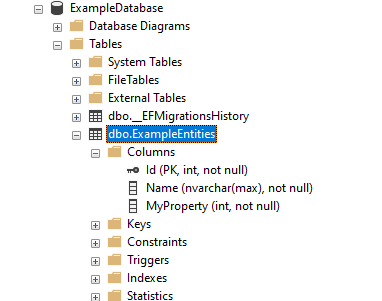


mig\_2’den mig\_1 migrationa geri gelmek için;

update-database mig\_1 talimatını kullanırız;

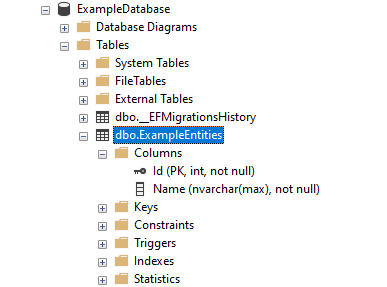
İlgili migrationdan istediğimiz migrationa güncellemek için:

updata-database [MigrationName] talimatını kullanırız.

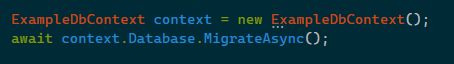


En son mig\_2 migrate edilmişti haliye veritabanımız üstteki gibi şimdi gelelim mig\_1 adlı migrationa geri çekelim.

update-database mig\_1 talimatını verdiğimizde veritabanımız mig\_1 deki değişiklikleri ele alarak veritabanını güncelleyecektir.



Run time da veritabanını migrate etmemiz gereken durumlar olabilir. Bu durumda DbContextimizden bir instance oluşturup, instance üzerinden Database propertysi üzerinden MigrateAsync Fonksiyonunu çalıştırmamız yeterlidir.



OutputType ClassLibrary olmamalı

OnConfiguring Fonksiyonu:

* EfCore Toolunu yapılandırmak için kullanılan bir methoddur.
* Context nesenesinde override edilerek kullanılır
* Hangi sunucudaki veritabanına hangi kullanıcı bilgileri ile bağlanacaksak bunun gibi temel konfigürasyonları belirlediğimiz methoddur.
* Bu fonksiyon içerisinde aşağıdaki yapılandırmalar gerçekleştirilir
  + - Provider
    - ConnectionString
    - LazyLoading
    - Vb.

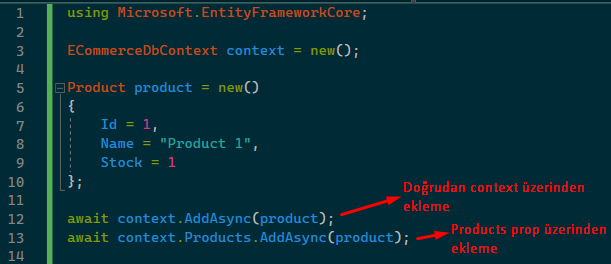
Basit Düzeyde Entity Tanımlama Kuralları:

* EfCore, her tablonun varsayılan olarak bir primary key kolonu olması gerektiğini kabul eder
* EfCore, Property adı Id,ID veya [EntityAdı]Id,[EntityAdı]ID olan propertyi primary key olarak kabul eder.

Veri Nasıl Eklenir?

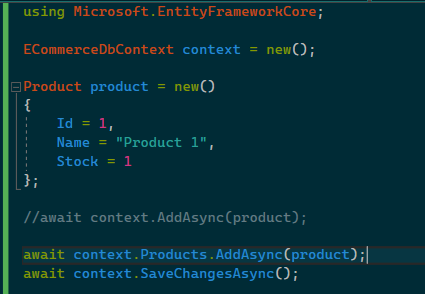
* CRUD işlemlerini yapabilmek için EFCore’da context sınıfından bir instance’a ihtiyaç vardır.
* Eklenecek olan verinin model sınıfından bir instancea ihtiyaç vardır
* Model olarak oluşturduğumuz her classın instanceı o veritabanı tablosunun 1 satırına denk gelir.
* İki şartıda sağladıktan sonra ekleme yapmak için context sınıfından oluşturduğumuz instance ile ilgili fonksiyonları çağırabiliriz
* Ya da tip güvenli çalışmak istiyorsak context sınıfından oluşturdğumuz instance ile DbSet propertylerine erişerek ilgili tabloya ekleme yapabiliriz.



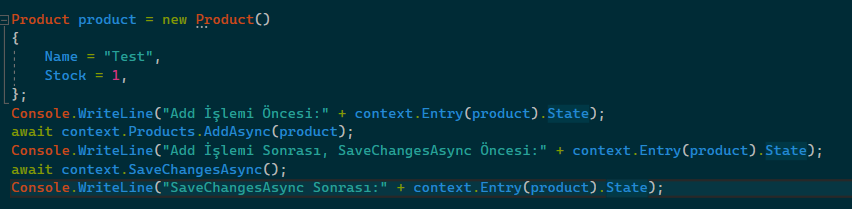


Eklediğimiz veri bu durumda kaydedilmeyecektir. Bunun için context nesnesi üzerinden SaveChanges methodunu çağırmamız gerekir.

SaveChanges methodu insert,update,delete gibi sorguların oluşturulup transaction eşliğinde veritabanına gönderip execute edilmesini sağlayan fonksiyondur.



EFCore Açısından Bir Verinin Eklenmesi Gerektiği Nasıl Anlaşılıyor?



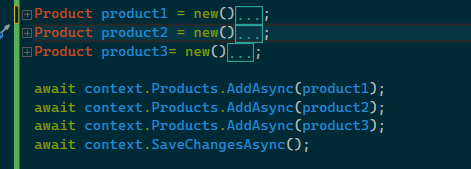


* Add işlemi öncesi: Veri üzerinde herhangi bir işlem yapılmadığı için veri bağımsız/ayrılmış olarak tanımlanır
* Add işlemi sonrası save öncesi: Eklenmeye hazır veri haline getirilmiş gibi düşünebiliriz.
* Save sonrası: İlgili veri veritabanına eklendiği sonucuna varabiliriz.

Birden Fazla Veri Ekleme

Veritabanı işlemlerinde transaction bir maliyettir. Veritabanı işlemleri yaparken ilgili işlemi en az transaction maliyeti ile yapmaya çalışmalıyız.

Haliyle her ürün eklendiğinde-silindiğinde-güncellendiğinde tekrar tekrar savechanges ile transaction açmak yerine tek bir transaction ile bu işlemi yapmalıyız.



Yukardaki işlemde tüm veriler eklendikten sonra saveleyerek tek bir transaction ile veritabanına gönderiyoruz.

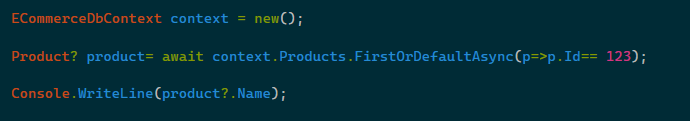
İlgili verileri tek tek eklemek yerine AddRange fonskiyonunu kullabailiriz. [AddRangeAsync(product1, product2, product3)]

Veri Nasıl Güncellenir?

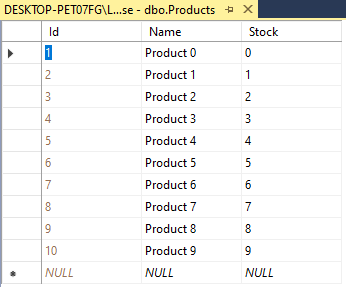
Bir verinin güncellenebilmesi için öncelikle ilgili verinin elde edilmesi gerekmektedir.

FirstOrDefault fonksiyonu: verdiğimiz şarta uygun verilerin ilkini getirir (daha sonra detaylı incelenecek)

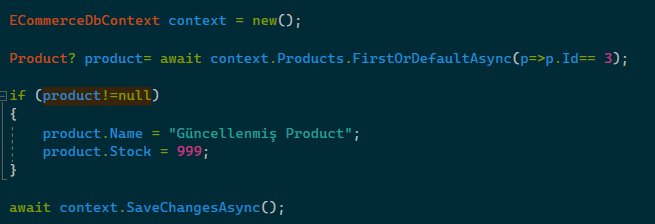
Context nesnemiz üzerinden Products propertysine erişerek FirstOrDefault fonksiyonu ile veritabanımızda bir sorgu oluşturup eğer varsa o veriyi elde edebiliriz.



Idsi 123 olan bir Product varsa getirecek. Eğer nullable(?) kullanmazsak bize exception fırlatacaktır.



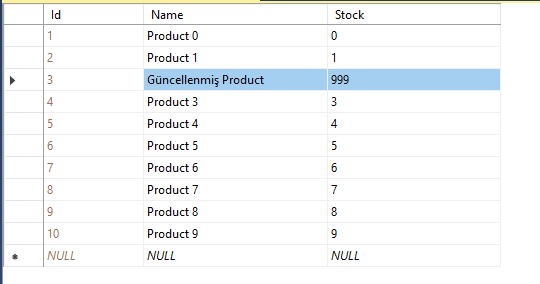
Güncelleme öncesi veritabanı yukarıdaki gibidir.



Elde ettiğimiz veri eğer boş değil ise güncelleme işlemini elde ettiğimiz veri üzerinden yapıyoruz.

Daha sonra SaveChanges ile veritabanımızı güncelliyoruz.

Güncelleme sonrası veritabanı aşağıdaki gibidir:

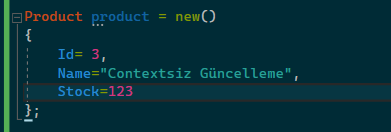


Change Tracker Nedir?

Change Tracker context üzerinden gelen verilerin takibinden sorumlu yapılanmadır. Bu yapılanma sayesinde nesnelerin delete veya update sorgularının oluşturulup oluşturulmayacağı anlaşılır.

Güncelleme kısmında olan kodlara bakarak:

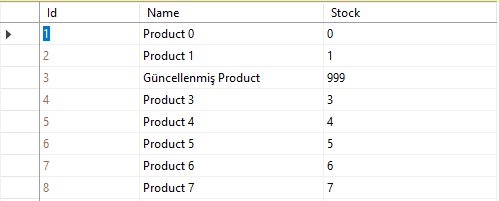
Id=3 olan veriyi çektiğimizde change tracker veriyi her sürekli olarak takip eder. Eğer verinin herhangi bir biriminde değişiklik olursa o verinin güncelleneceği veya silineceği kaanitine varıyor.

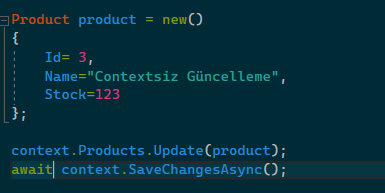


Veritabanı üzerinde Idsi 3 olan veriyi bu şekilde güncellemek istediğimzde ilgili veriyi context üzerinden elde etmediğimiz için track (izlemek-takip etmek) edilmeyecek haliyle Change Tracker devreye girmeyecektir. Dolayısıyla takip edilmeyen bir nesneyi güncellemiz gerektiği senaryolarda EntityFrameworkCore tarafından Update Fonskiyonu üretilmiştir.

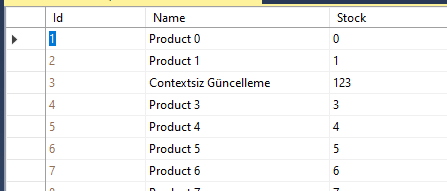
Kısaca track edilmeyen nesneleri güncellemek için yine context üzerinden ilgili dbset property ile Update fonksiyonunu çağırarak güncelleme yapabiliriz

Güncelleme yapılmadan önce veritabanı:





Güncelleme yapıldıktan sonra veritabanı:



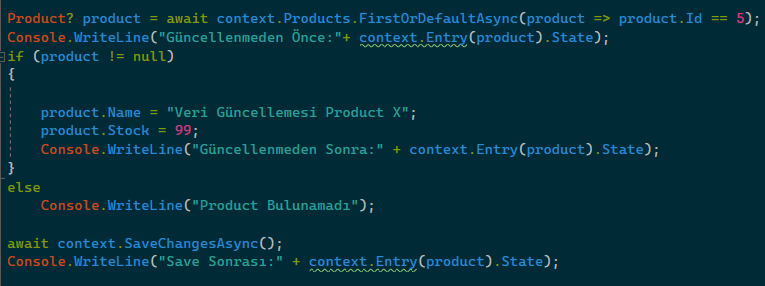
Entity State Nedir?

Bir entity instancenın durumunu ifade eden referanstır. İlgili nesnenin veritabanında yapılacak olan işlemini yakalamamıza yardımcı olur.

[EFCore Açısından Bir Verinin Eklenmesi Gerektiği Nasıl Anlaşılıyor?

Başlığını incele]

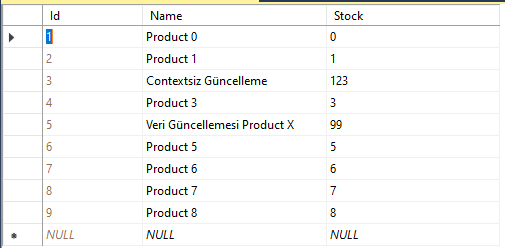
EFCore Açısından Bir Verinin Güncellenmesi Gerektiği Nasıl Anlaşılıyor?

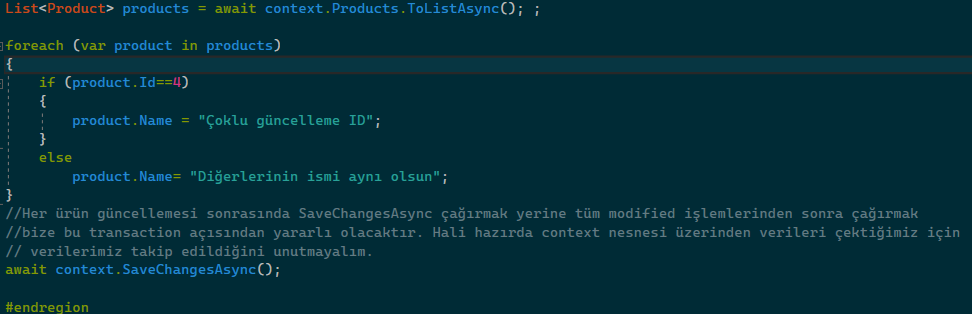




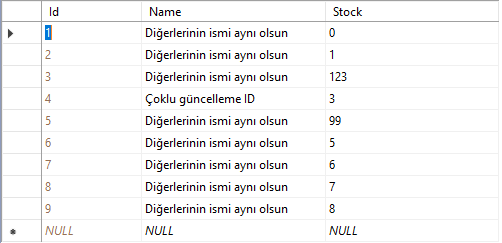
Eklemede olduğu gibi çoklu güncellemelerde de tüm güncellemelerden sonra SaveChanges çağırmak tek bir transaction ile tüm işlemi bitirmemizin yanında bize maliyet açısından bir artı kazandıracaktır.

Güncelleme yapılmadan önce veritabanı:



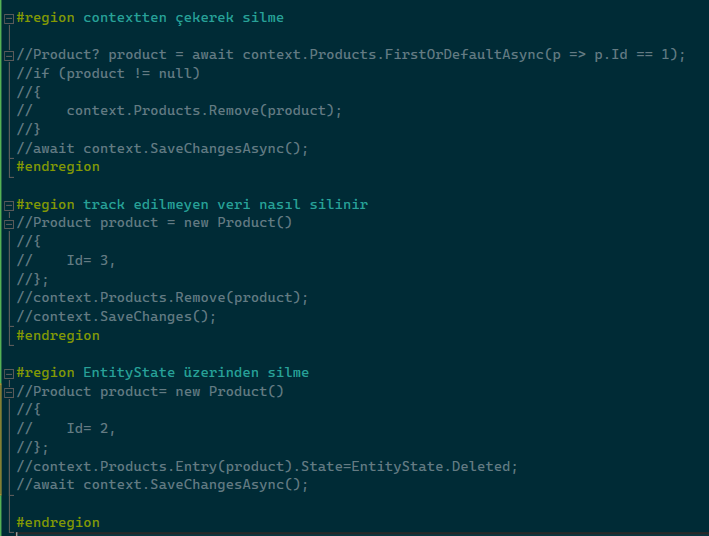


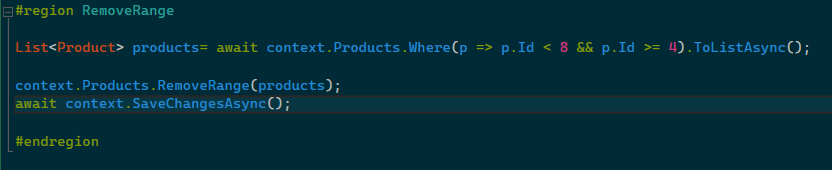
Güncelleme yapıldıktan sona veritabanı:



Veri Nasıl Silinir?

Güncelleme işleminde olduğu gibi yine aynı şekilde işlem yapılacak olan veriyi veritabanından çekip gerekli işlemleri yaptıktan sonra SaveChanges fonksiyonu çağrılarak işlem sonlandırılır. Silme işleminde EntityState Delete olarak işlenir. Yine güncelleme işleminde olduğu gibi savechanges fonskiyonunu verimli kullanarak [her işlem(update-delete) sonrasında değilde tüm işlemler bittikten sonra bize trasactiondan tasarruf etmemize yardımcı olacaktır].

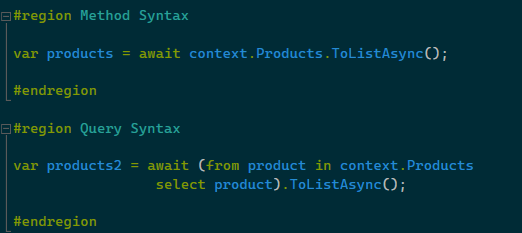




Temel Düzeyde Sorgulama Yapılanmaları

Method Syntax: Sorgulama yaparken methodlar kullanılıyorsa method syntax denir.

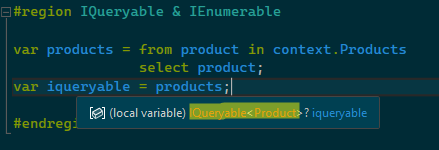
Query Syntax: Sorgulama yaparken queryler kullanılıyorsa query syntax denir.



Method syntax da context nesnesinin propertyleri üzerinden methodlar çağrılırken query syntaxda linq komutları oluşturularak methodlar çağrılır.

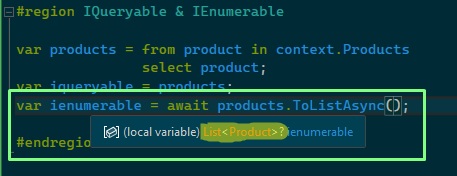
Temel olarak IQueryable ve IEnumarable Nedir?

IQueryable: Ham sorguya denk gelir. Yani oluşturulmuş sorgunun execute edilmemiş halidir. (Elimizde veri yok)



IQueryable içerisinde ilgili türün instancelarını barındırır(döngü).

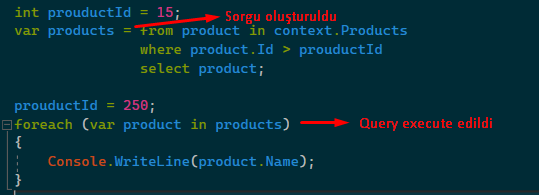
IEnumarable: Sorgunun execute edilmiş halidir. Yani veriler sorgu sonucunda elde edilen veriler inmemorye yüklenmiştir. (Elimizde veri var)



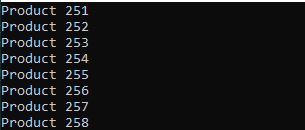
Burada tolist fonksiyonu sorguyu execute eder.

Deffered Execution(Ertelenmiş Çalışma)

IQueryable sorgularında ilgili çalışma yazıldığı noktada sorguyu generate etmez. Sorgu execute edildiği noktada sorgu oluşturulur.

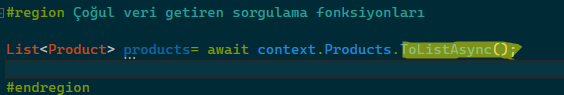


Sorgu execute edilmeden önce productId 250 olduğu için 250 ye göre arama yapacaktır.



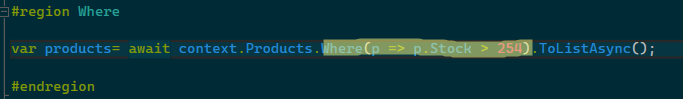
Çoğul Veri Getiren Sorgulama Fonksiyonları:

ToList: Üretilen sorguyu execute eder.

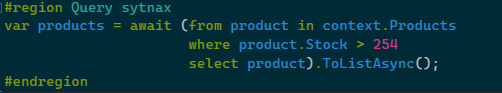


Where Fonksiyonu: Oluşturulan sorguya where şartı eklememizi sağlar

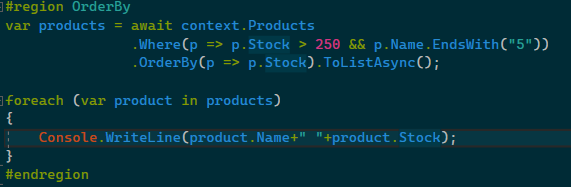
Method syntax:

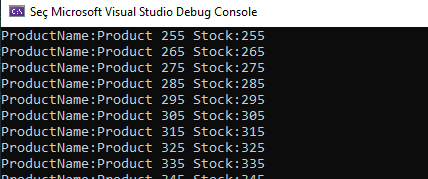


Query syntax:

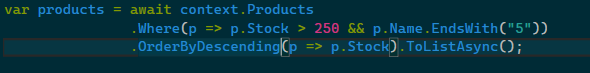


OrderBy Fonksiyonu: Sorgu üzerinde sıralama yapmamızı sağlayan fonksiyondur.



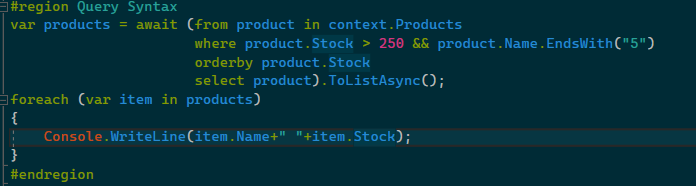


OrderBy metodu default olarak artan(ascending) olarak sıralar ancak OrderByDescending metodu ile azalan olarak sıralayabiliriz.

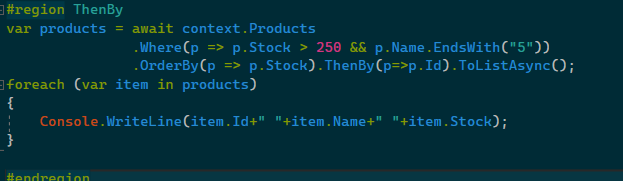




Yine bu fonksiyonu da query syntax olarak kullanabiliriz.



ThenBy Fonksiyonu: Ordery üzerinde yapılan sıralamaya farklı kolonlara da uygulamamızı sağlayan fonksiyondur.



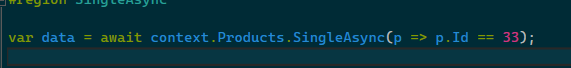
Eğer ki Stock aynı olanlar var ise bu seferde idye göre sıraya sokacaktır.

Tekil Veri Getiren Sorgulama Fonksiyonları

Single Fonksiyonları: Temel amaç sorgu sonucunda tek bir veriyi getirmektir.

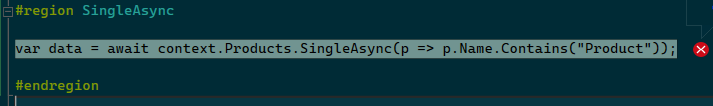
Single: Sorgu sonucunda birden fazla veri geliyorsa ya da hiç veri gelmiyosa exception fırlatır.

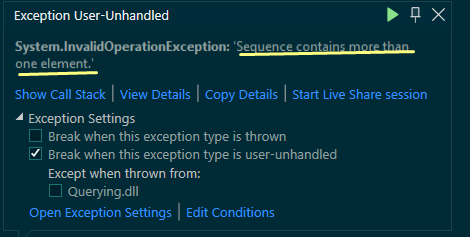
SingleOrDefault: Sorgu sonucunda birden fazla veri geliyorsa exception fırlatır hiç veri gelmiyorsa null döner.



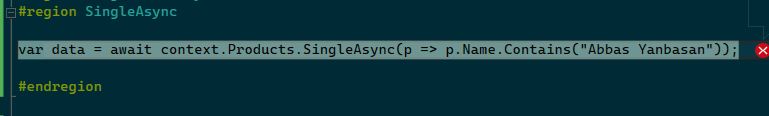
Sorgu sonucunda birden fazla veri geliyorsa ya da hiç veri gelmiyosa exception fırlatır.

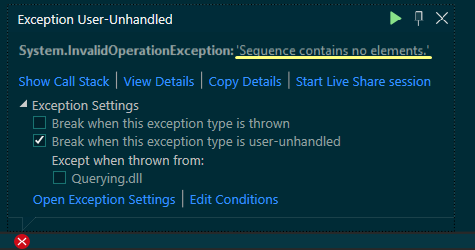
Birden fazla veri olma durumu:



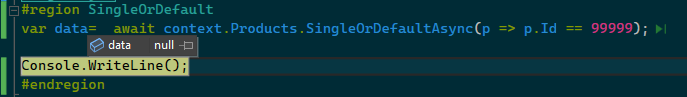


Hiç verinin olmaması durumu:

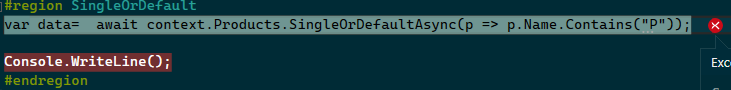


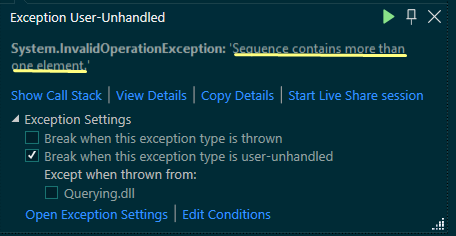


SingleOrDefaultta hiç veri gelmiyorsa null döner.



Eğer birden fazla veri geliyorsa exception fırlatır.





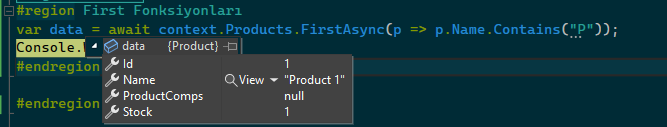
First Fonksiyonları: Temel amaç tek bir veriyi elde etmektir. Single fonksiyonuyla farkı Single da birden fazla veri barındırma durumunda hata verir ancak first fonskiyolarında bulduğu ilk veriyi getirir birden fazla olup olmaması durumu değiştirmez ancak verinin bulunamaması durumunda hata verir.

Örneğin veritabanında Ahmet isimli kullanıcıları çekmek istiyoruz sorgu sonucunda First ve Single fonksiyonlarını inceleyelim.

First fonksiyonu sonucunda istek tek veri olsun ister birden fazla olsun içlerinden ilkini bize dönderir. Verinin bulunamaması durumunda hata dönderir.

Single fonskiyonu durumunda eğer Ahmet isminde birden fazla kullanıcı varsa ya da hiç yoksa bize hata dönderecektir.

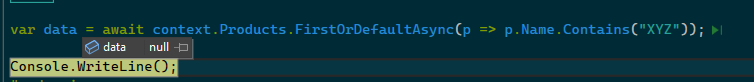
Birden fazla verinin olma durumu:



Hiç verinin olmama durumu:



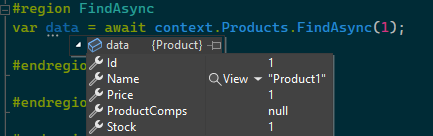
Eğer Fonksiyonumuz FirstOrDefault ise hata fırlatmayacak null dönecektir.



First vs Single



Find Fonksiyonları: Find fonksiyonu primary key kolonuna özel hızlı bir arama yapmamızı sağlayan fonksiyondur.





Last Fonksiyonu: Sorgu sonucunda gelen verilerden en sonuncusunu bize getirir. OrderBy kullanılması mecburidir.

Aşağıdaki fonskiyonlarda IQueryablede çalışmak maliyeti daha da düşürür. IEnumrableda çalışmak maliyetlidir. Bu fonskiyonlar sorguyu execute eder.

Count Fonksiyonu: Oluşturulmuş sorgunun execute neticesinde kaç tane veri geleceğini bize bildiren fonksiyondur(int).

LongCount Fonksiyonu: Big datada çalışırken oluşturulmuş sorgunun execute neticesinde kaç tane veri geleceğini bize bildiren fonksiyondur(long).

Any Fonksiyonu: Sorgu neticesinde verinin gelip gelmediğini veren fonksiyondur(bool).

Max Fonksiyonu: Verdiğimiz şarta bağlı olarak bize şarttaki max değerini dönderir.

Min Fonksiyonu: Max fonskiyonunun min halidir.

Distinct Fonksiyonu: Sorguda tekrar eden kayıtlar varsa bunları tekilleştiren bir işleve sahip fonksiyondur. IQueryable döner execute edilmesi gerekir.

All Fonksiyonu: Yapılan sorgu sonucunda gelen verilerin, verilen şarta uyup uymadığını kontrol eder. Eğer uyuyorsa true uymuyorsa false döner.

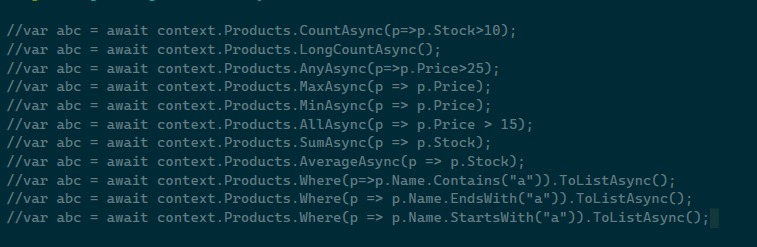
Sum Fonksiyonu: Vermiş olduğumuz sayısal propun toplamını alır.

Avarage Fonksiyonu: Vermiş olduğumuz sayısal propun aritmetik ortalamasını alır.

Contains: Like sorgusu oluşturmamızı sağlar. Where de kullanmamız gereklidir. ‘%...%’

StartWith: Like sorgusu oluşturmamızı sağlar. Where de kullanmamız gereklidir.’…%’

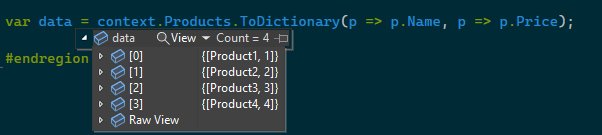
EndWith: Like sorgusu oluşturmamızı sağlar. Where de kullanmamız gereklidir.’%...’



Sorgu Sonucu Dönüşüm Fonskiyonları:

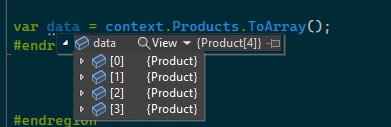
Bu fonksiyonlar ile sorgu sonucunda gelen verileri istediğimiz türlerde elde edebiliriz.

ToDictionary: Sorguyu execute eder ancak gelecek olan veri dictionary formatındadır.



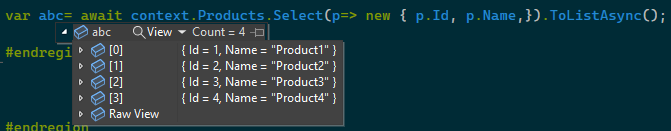
ToList ile aynı mantıkta çalışır fakat ToList bize List<Entity> türünde değer dönderirken ToDictionary bize Dictionary<Key,Value> türünden değer dönderir.

ToArray: Yine ToList ile aynı mantıkta çalışır ancak entity türünden listte değil arrayde tutat. (Entity[]).



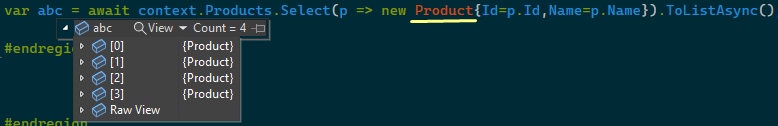
Select: Select fonksiyonunun birden fazla davranışı söz konusudur. Sorgu execute edilmeden önce kullanılmalıdır.

1.Davranış: Oluşturulacak olan sorguda belirli kolonları sorgulamıza, bu ayarları yapmamızda bize yardımcı olur.

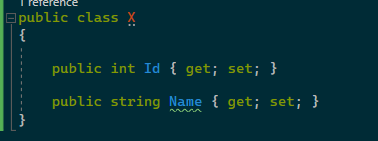


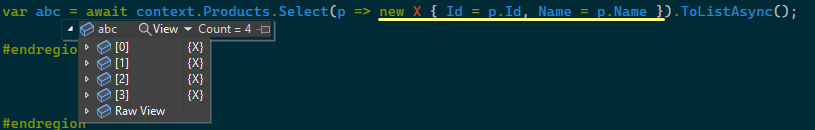
2.Davranış: Gelen verileri farklı türlerde karşılamamızı sağlar.

1.Davranışdaki veri türü anonim bir türdür. Altta ise Producttır.

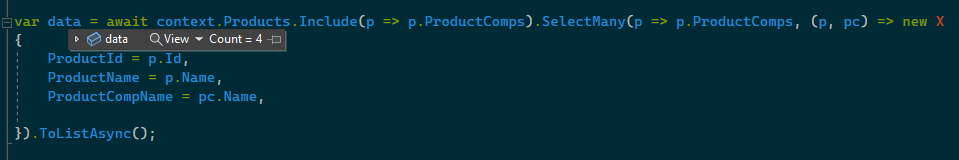


İstediğimiz veri türünü karşılayacak propertyleri barındıran bir class oluşturarak gelen datayı karşılayabiliriz.



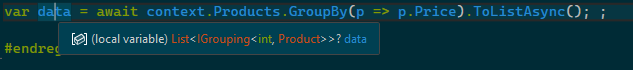


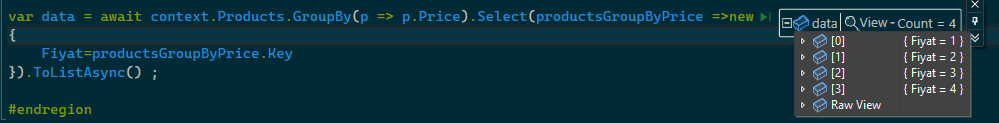
SelectMany: Select ile aynı amaca hizmet eder ancak, ilişki kurduğumuz tablolar içerisindeki kolonları da ilgili sorguya eklemek istiyorsak SelectMany kullanmalıyız.



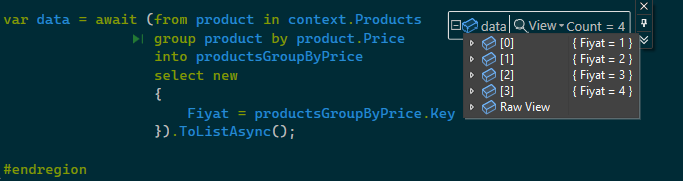
Gruplama Fonksiyonu

Gruplama yapmamızı sağlayan fonksiyondur.



Burda elde edilen verileri almak için aşağıdaki davranışı sergilemeliyiz;  


Query syntax:

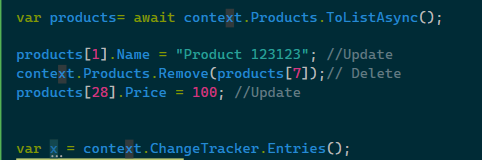


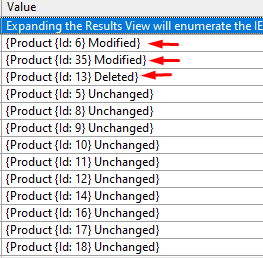
Change Tracker

Context nesnesi üzerinden çağrılan veriler bir takip mekanizması tarafından takip edilir. Bu mekanizmaya Change Tracker denir. Change Tracker nesneler yani veriler üzerinde yapılan işlemler sonucunda uygun sql sorgularını generate eder. Bu işleme Change Tracking denir.

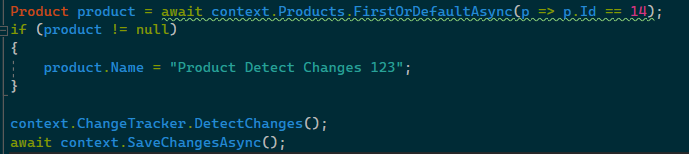


Change Tracker Propertysi: Takip edilen nesnelere eriştiğimiz ve gerektiği takdirde işlemler yapabildiğimiz proptur. DbContext içersinde tutulur.





DetectChanges Metodu: Veri üzerinde yapılan değişikliklerin, ChangeTracker mekanizması tarafından yakalandığından emin olmak için kullanırız. Her zaman kullanılması lüzumsuzdur ancak bilmekte fayda var. SaveChanges ve Entries fonksiyonları içerisinde bu metodu çağırır.

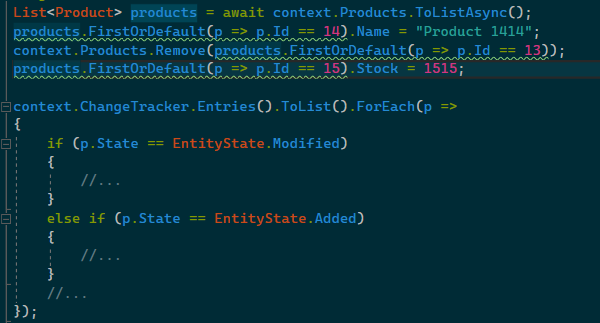


AutoDetectChangesEnabled Prop: İlgili metodlar (Entries, SaveChanges) tarafından DetectChanges metodunun otomatik olarak çağrılmasınının konfigürasyonunu yapmamızı sağlar. Bu özellik bazı durumlarda kapatılarak bize artı performans eksi maliyet kazanadırır.

Entries Metodu: Contextdeki Entry metodunun Collectionlar üzerinde çalışmamızı sağlayan halidir.

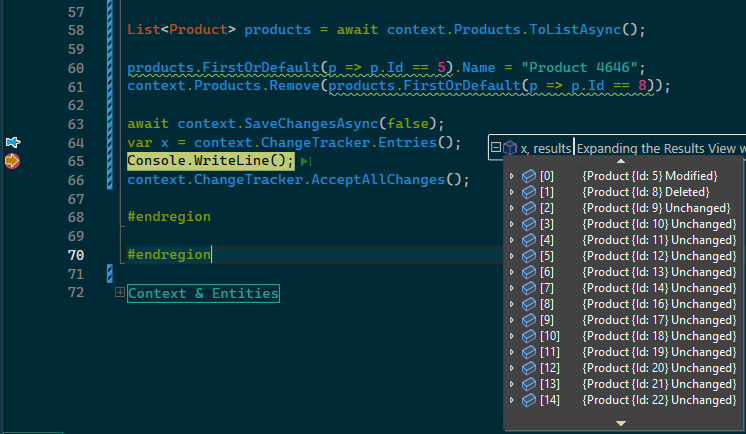
ChangeTracker mekanizması tarafından izlenen her entity nesnesinin bilgisini EntityEntry türünden elde etmemizi sağlar ve belirli işlemler yapabilmemize olanak tanır.

Entries metodu, DetectChanges metodunu tetikler. [AutoDetectCh.]

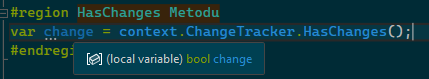


AccepAllChanges Metodu: SaveChanges() veya SaveChanges(true) çağrıldığında EFCore takip ettiği nesneleri track etmeyi bırakacağı için, ilgili operasyonda herhangi bir hata meydana geldiğinde tekrar bir düzeltme söz konusu olmayacaktır.

Bu durumda SaveChanges(false) metodu ile nesnelerin sürekli olarak track edilmesini sağlarız. İlgili operasyonun başarılı olduğundan herhangi bir hata meydana gelmediğinden emin olarak AcceptAllChanges metodu ile nesnelerin track edilmesini kesebiliriz.



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



EntityStates: Entity nesnelerinin durumlarını ifade eder.

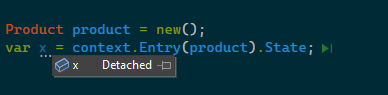
Detached: Nesnenin change tracker tarafından takip edilmediğini ifade eder.

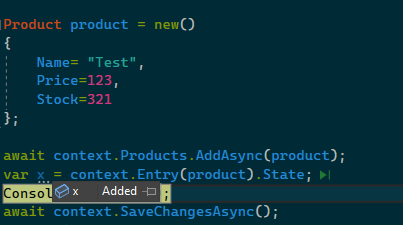
Added: Veritabanına eklenecek olan ancak henüz işlenmemiş olan veriyi ifade eder. SaveChanges ile veritabanına insert edilir.

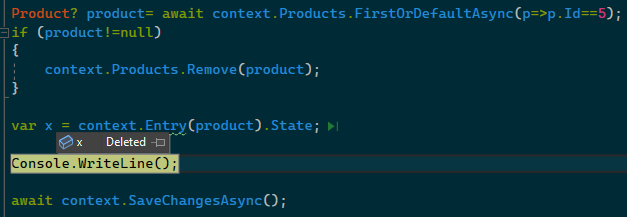
Deleted: Veritabanından silinecek olan ancak henüz işlenmemiş olan veriyi ifade eder. SaveChanges ile veritabanından silinir.

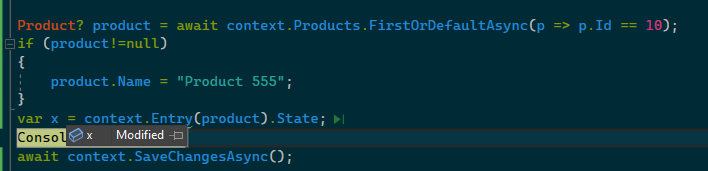
Modified: Veritabanına var olan güncellenecek ancak henüz işlenmemiş olan veriyi ifade eder. SaveChanges ile veritabanına insert edilir.

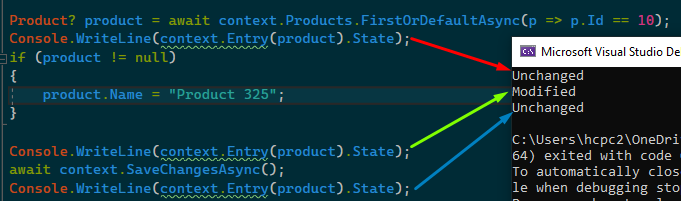
Unchanged: Veritabanınından sorgu neticesinde çekilen ancak üzerinde herhangi bir değişiklik yapılmayan nesneyi ifade eder.





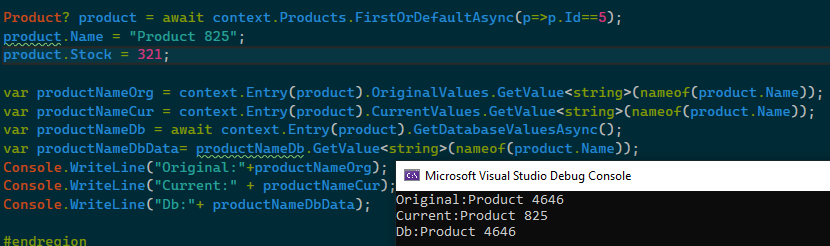






Entry Propertyleri

* OriginalValues: İşlem yapılan verinin orijinal verilerini yani veritabanındaki verilerini elde etmek için kullanılır. GetDatabaseValues propertyside bu işlem için kullanılabilir.
* CurrenValues: İşlem yapılan verinin o anki verilerini barındıran propertydir.



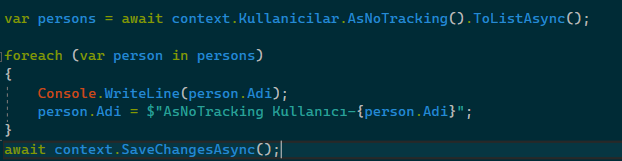
NOT: SaveChanges metodu virtual bir metoddur Context sınıfımız içerisinde override edilerek ChangeTrackerı interceptor olarak kullanabiliriz.

AsNoTracking, AsNoTrackingWithIdentityResolution, UseQueryTrackingBehavior

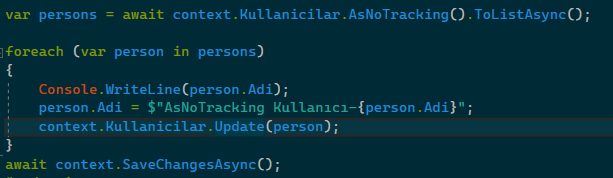
AsNoTracking: Context üzerinden gelen tüm datalar Change Tracker ile takip edilmektedir. Haliyle gelen her verilerin boyutuyla orantılı maliyeti vardır. Eğer gelen veriler üzerinde herhangi bir işlem yapılmayacaksa amacımız verileri listelemek ise bu verilerin track edilmesi bize ekstra maliyete sebeb olacaktır.

AsNoTracking metodu ile gelen verilerin takip edilmemesini sağlarız. Sonuç olarak veriler track edilmez ve track sonucunda oluşacak maliyetler engellenir.

AsNoTracking fonksiyonu ile yapılan sorgulamarda veriler elde edilir ancak elde edilen veriler üzerinde işlem yapılamaz.

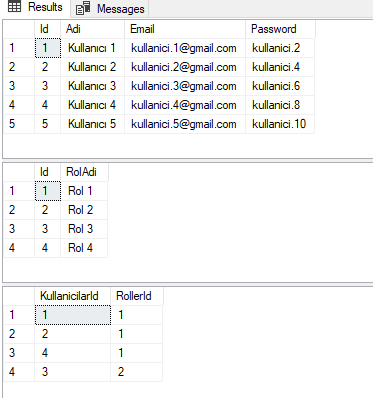


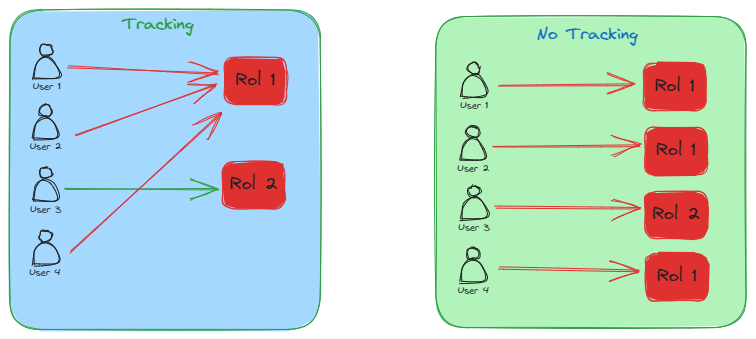
ChangeTracker devre dışı olduğundan herhangi bir değişiklik yapılmayacaktır.



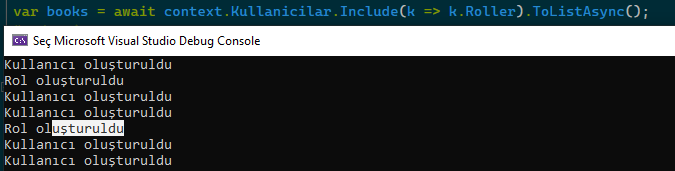
Context üzerinden dbset propu ile update fonksiyonuyla track edilmeyen nesneleri güncelleyebiliriz.

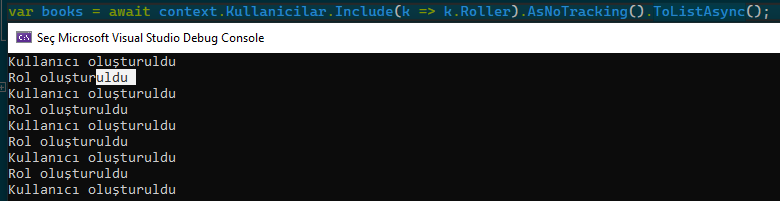
AsNoTrackingWithIdentityResolution: İlişkisel tablolarda sorgula-ma yaparken AsNoTracking yinelenen dataların herbirine ayrı bir instance oluşturur.



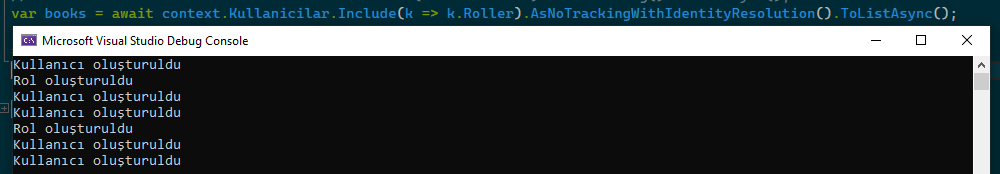


Tracking ve NoTracking durumlarında yukardaki resimdeki gibi olacaktır. Tracking durumunda 7 nesne dönerken No Tracking durumunda 9 nesne dönecektir.





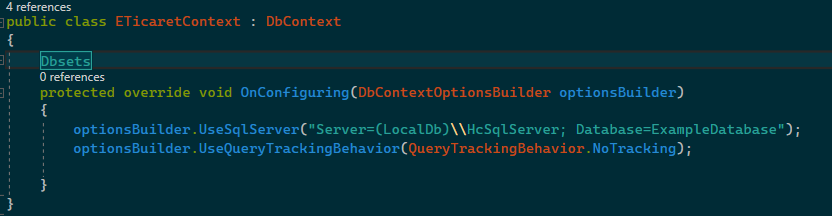
Böyle bir durumda AsNoTrackingWithIdentityResolution metodunu kullanmak performans açısından sağlıklıdır.



AsNoTrackingWithIdentityResolution, AsNoTracking’e göre daha maliyetli ancak ChangeTracker’a göre daha az maliyetlidir.

AsTracking: ChangeTracker’ı iradeli şekilde devreye sokmamıza yarayan fonksiyondur.

UseQueryTrackingBehavior: Uygulama seviyesinde context üzerinden gelen dataların ChangeTracker mekanizmasının davranışını ayarlamızı sağlar yani konfigürasyon fonksiyonudur.



Yukardaki yaptığımız konfigürasyonda artık contexten gelen datalar change tracker ile takip edilmeyecektir. Haliyle track edilmesi gereken noktalarda AsTracking metodunu çağırmamız gerekmektedir.

İlişkisel Yapılar

Terimler;

Principal Entity(Asıl Entity): Herhangi bir tabloya bağımlı olmayan yani içerisinde farklı tablolardan alan barındırmayan, tek başına var olabilen entitydir.

Dependent Entity(Bağımlı Entity): Herhangi bir tabloya bağımlı olan yani içerisinde farklı tablolaların alanlarını barındıran entitydir. Kendi başına var olamaz.

Foreign Key: Principal entity ile Dependent entity arasındaki bağlantıyı sağlayan keydir. Dependent entityde tanımlanır. Dependent entity içerisinde principal keyi tutar.

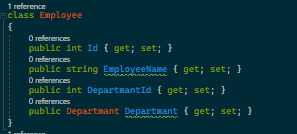
Principal Key: Principal entitydeki idnin kendisidir.

|  |  |
| --- | --- |
| Employees   * Id * EmployeeName * DepartmandId | Departmans   * Id * DepartmanName |

Yukardaki tabloda Principal Entity Departmans tablosunu modelleyen Deparmant entitysidir. Dependent Entity ise Employees tablosunu modelleyen Employee entitysidir. Bu ikisi arasında bağlantıyı sağlayan Foreign key ise DepartmanIddir.

Navigation Property: İlişkisel tablolar arasındaki fiziksel erişimi sağlayan tablo modelleri(entity) üzerinden sağlayan propertylerdir.

Yukardaki tabloda 1-n bir ilişki vardır. Yani bir employee bir departmana sahipken bir departmanın birden fazla employeesi olabilir. Bu durumda Employee ve Departman entitylerinde birbirleri üzerinden erişim sağlamak için propertyler tanımlanır. Tanımlanan propertyler ilgili entity türünden olmalıdır.



İlişki Türleri;

One to One: 1-1 olan ilişkidir. Karı-koca ilişkisi örnek verilebilir (Medeni Hukuk).

One to Many: 1-n olan ilişkidir. Anne-çocuk ilişkisi örnek verilebilir.

Many to Many: n-n olan ilişikidir. Çalışanlar-Projeler ilişkisi örnek verilebilir.

İlişki Yapılandırma Yöntemleri:

Default Conventions: Varsayılan entity kuralları kullanarak yapılan ilişki yapılandırma yöntemidir. Navigation propertyler üzerinden EFCore’un yaptığı yapılandırma denilebilir.

Data Annotations Attributes: Entitynin niteliklerine göre ince ayarlar yapmamızı sağlayan attributelardır. Örneğin DepartmandId yi “X” adında tutmak istiyorsak bu propertye Data Annotation Attribute bildirmemiz gerekmektedir (Foreign Key).

Fluent API: Entity modellerindeki ilişkileri yapılandırırken daha detaylı çalışmamızı sağlayan yöntemdir.

HasOne: İlgili entitynin ilişkili olduğu entitye birebir ya da bire çok olacak şekilde ilişkisini yapılandırmaya başlayan metoddur.

HasMany: İlgili entitynin ilişkili olduğu entitye çoka bir ya da çoka çok olacak şekilde ilişkisini yapılandırmaya başlayan metoddur.

WithOne: HasOne ya da HasManyden sonra hangi ilişki türünde olacaksa (HasOne seçilmiş 1-1 HasMany seçilmiş ise n-1) ilişki yapılandırmasını tamamlayan metoddur.

HasOne🡺WithOne 🡺 1-1

HasMany🡺WithOne🡺 n-1

WithMany: HasOne ya da HasManyden sonra hangi ilişki türünde olacaksa (HasOne seçilmiş 1-n HasMany seçilmiş ise n-n) ilişki yapılandırmasını tamamlayan metoddur.

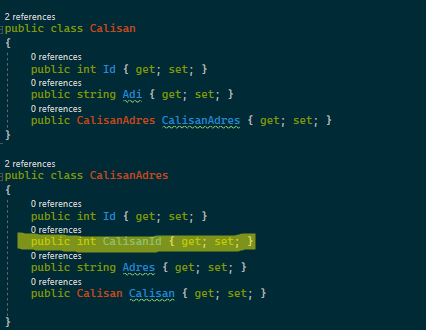
HasOne🡺WithMany 🡺 1-n

HasMany🡺WithMany🡺 n-n

One To One: Birebir ilişki kurarken öncelikle entityler üzerinde navigation propertylerin belirlenmesi gerekmektedir

Default Convention:

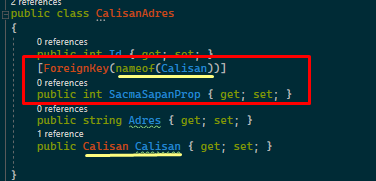
Eğer default conventionsda çalışıyorsak migrate ederken EfCore hangi entitynin dependent hangi entitynin principal entity olduğunu anlamadığından foreign key belirtmemiz gerekmektedir.



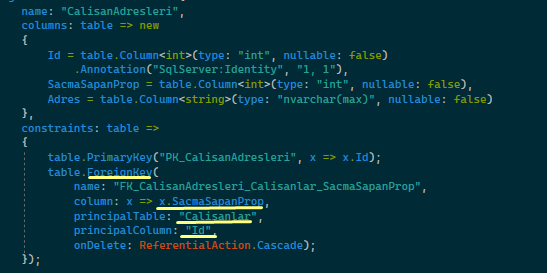
Her iki entityde birbirlerini tekil olarak ifade edeceğini navigation propertyler ile belirlenmiştir. Ayrıca dependent entitynin belirlenmesi için ilgili property oluşturulmuştur.

Data Annotations:

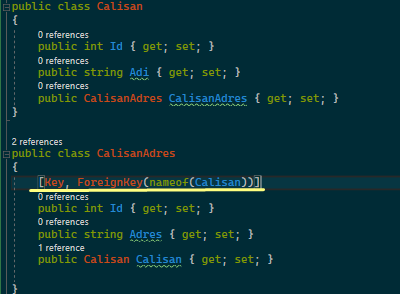
Default Conventions da dependent entityi belirtmek için x\_Id olarak isimlendirme yapmamız gerekiyordu ancak DA’da Foreign Key attribute ile ilgili entitynin navigation propertysini belirterek herhangi bir isimdeki property oluşturabiliriz.



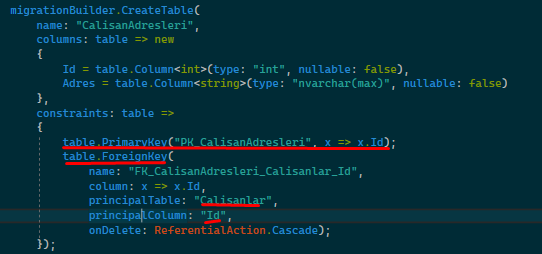
Migrate sonucunda oluşan migrationda ilgili propertynin foreign key olarak eklendiğini göreceğiz.



Bu işlem sonucunda hem extra bir kolon oluşturmak zorundayız hemde veritabanında index tutulması gerekmektedir. Haliyle bir maliyet söz konusu olduğundan aşağıdaki işlem bu maliyetlere girmememizi sağlar ve 1-1 olan ilişkimizi garanti altına alır.

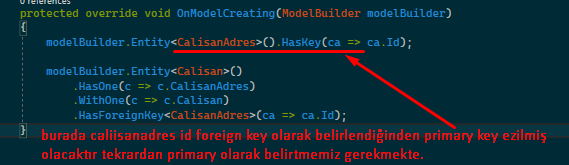


Migrationa baktığımızda Id kolonu hem PK hemde Calisan Idsini temsil eden FK olacaktır.



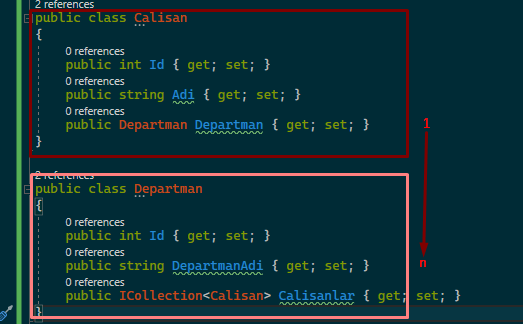
Yine DA yönteminde de navigation propertyler tanımlanmalıdır ayrıca foreign key için bir property tanımlamak zorunda olmadan üstteki işlemdeki gibi foreign key tanımlayabiliriz.

Fluent API:

 Fluent apida çalışıyorsak entityler üzerinde değilde context context içerisinde onModelCreating fonksiyonunda çalışırız.

One To Many

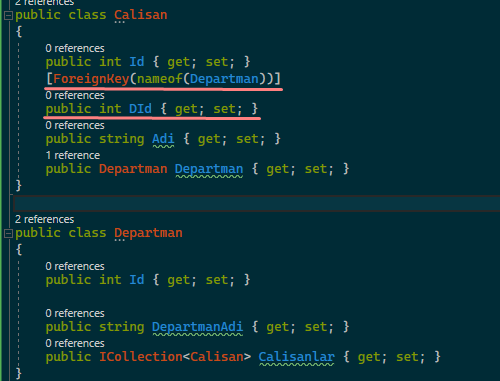
Default Convention:



Default conventionda foreign key kolonu oluşturmak zorunda değiliz oluşturmadığımız takdirde EfCore kendiliğinden oluşturacaktır.

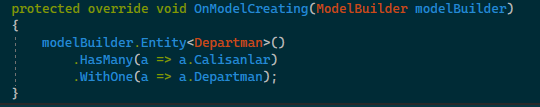
Data Annotation:

Eğer ki foreign key kolonuna denk gelen propertye bizim oluşturduğumuz özel isimin verilmesini istiyorsak artık DA kullanmalıyız.

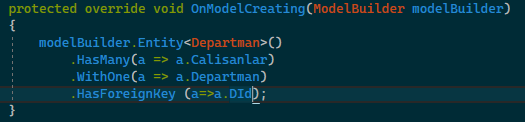


Fluent API:

FAPI da çalışırken onModelCreating fonksiyonu üzerinde çalışırız.

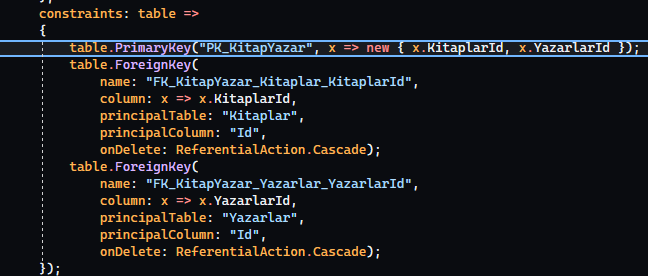


Eğer foreign keye özel isim vermek istiyorsak;



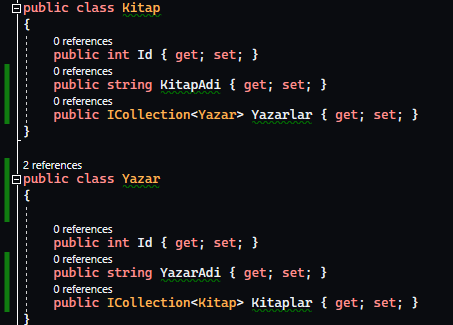
Many To Many

Çoka çok ilişkide bir ara tablo oluşması söz konusudur bu oluşturulan tabloda ilişki içerisinde bulunan tabloların primary keyleri üzerinden ilişki kurulur. Sonuç olarak her iki tablonunda keyleri composit key olarak primary key olarak tutulur.



Üstte görüldüğü gibi hem kitaplar tablosundaki idyi hemde yazarlar tablosundaki idyi primary key olarak almıştır. Ayrıca cross table oluşturmamıza gerek yoktur.

Default Convention:

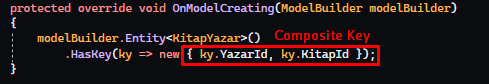


CrossTable aşağıdaki gibi oluşacaktır;

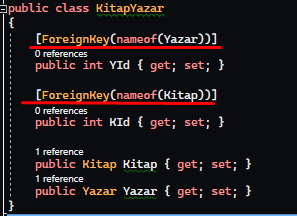


Data Annotations: Cross Table manuel olarak oluşturulmak zorunda dır. Cross table oluşturup ilgili entityler üzerinden 1-n olacak şekilde ilişki kurmalıyız. Bu ilişki sonucunda entityler kendi aralarında n-n bir ilişkiye sahip olacaktır. Composit key vermemiz için her iki entitynin idsine [Key] attribute vermemiz yeterli olmayacaktır. Haliyle burada Fluent API kullanmak zorundayız. Ayrıca oluşturulan cross table dbcontext içerisinde dbset olarak vermemize gerek yoktur.

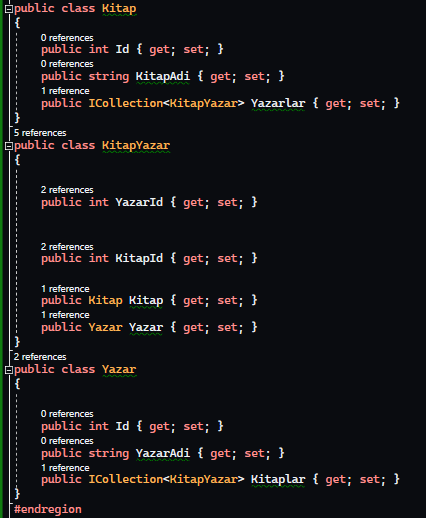


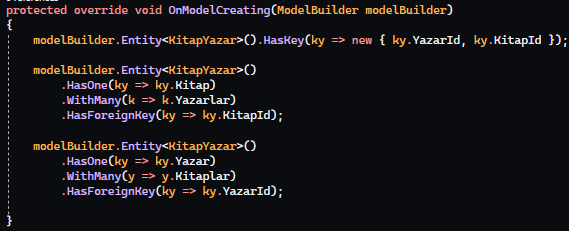


Eğer ki foreign keylere farklı isim vermek istiyorsak Foreign Key atrribute ile belirtmemiz gerekmektedir.



Fluent API: Yine bu yöntemde de cross table manuel olarak oluşturulmalıdır. Composite key haskey metodu ile uygulanmalıdır. Cross table dbset olarak verilmesine gerek yoktur.





CrossTable üzerinden entityler arası 1-n ilişki oluşturacağız haliyle entityler ile crostable arasında n-n ilişki sağlanmış olacaktır. Ayrıca foreign keyler belirtilmelidir.

İLİŞKİSEL TABLOLARDA VERİ EKLEME

1-1 İLİŞKİLERDE VERİ EKLEME:

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 ExampleDbContext:DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("ConnectionString");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

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

}

}

1.YÖNTEM – PRINCIPAL ENTITY UZERINDEN DEPENDENT ENTITY EKLEME

Person person = new();

person.Name = "Hüseyin Çoraklı";

person.Address = new() { PersonAddress = "Kahramanmaraş/Dulkadiroğlu" };

await context.People.AddAsync(person);

await context.SaveChangesAsync();

Principal entity eklerken, dependent entity eklemek zorunda değiliz çünkü principal bağımsız olan entitydir.

Ancak Dependent entity eklerken principal entityde eklemek zorunludur çünkü dependent entity principal entitynin varlığına bağımlıdır. Dependent üzerinden eklemede Efcore öncelikle principal entity ekleyecektir.

2.YÖNTEM – DEPENDENT ÜZERİNDEN PRINCIPAL EKLEME

Address address = new();

address.PersonAddress = "Kahramanmaraş/Pazarcık";

address.Person = new()

{

Name = "Mustafa Arı"

};

await context.Addresses.AddAsync(address);

await context.SaveChangesAsync();

Bu sorgu çalıştırıldığında EFCore öncelikle Principal entity olan Person oluşturacak ardından Dependent entity eklenecektir.

1-ÇOK İLİŞKİLERDE VERİ EKLEME

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 ExampleDbContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("ConnectionString");

}

}

1.YÖNTEM PRINCIPAL ÜZERİNDEN DEPENDENT EKLEME

Nesne üzerinden ve Object Initializer üzerinden ekleme olarak iki farklı ekleme işlemi vardır. Nesne üzerinden dependent entity eklemek için Principal entity constructor içerisinde dependent entity koleksiyonel olarak newlenmelidir aksi halde null geleceğinden dolayı program exception fırlatacaktır. Object Initilizaer üzerinden ekleme yapacak isek constructorda newlemeye gerek yoktur ancak ilgili constructorın içerisinde koleksiyonel olarak dependent entitynin newlenmesi daha sağlıklı ve gereklidir.

-Nesne Referansı Üzerinden Ekleme

Blog blog = new();

blog.Name = "Hacının Blogu";

blog.Posts.Add(new() { Title = "İttihad ve Terakki" });

blog.Posts.Add(new() { Title = "Cemal Paşa Kimdir?" });

blog.Posts.Add(new() { Title = "Enver Paşa Kimdir?" });

//bu yöntem için blog constructor da Post koleksiyonel yapıda newlenmeli

await context.Blogs.AddAsync(blog);

await context.SaveChangesAsync();

-Object Initializer Üzerinden Ekleme

Blog blog2 = new()

{

Name = "Mustafanın Blogu",

Posts = new HashSet<Post>() { new() { Title="Bitcoin Nedir?"},new() { Title= "Order Block Nedir?" } }

};

await context.AddAsync(blog2);

await context.SaveChangesAsync();

2.YÖNTEM DEPENDENT ÜZERİNDEN PRINCIPAL ENTITY EKLEME

Bu yöntemde sadece bir tande dependent entity eklenebilir bu yüzden bu yöntem tercih edilmemesi gerekir. Bazı senaryolarda bu durum gerekebilir.

Post post = new()

{

Title = "Nihilizm Nedir?",

Blog = new() { Name="Çağatayın Blogu"}

};

await context.Posts.AddAsync(post);

await context.SaveChangesAsync();

3.YÖNTEM FOREIGN KEY ÜZERİNDEN EKLEME

Üstteki durumda pricipal entity önceden var olması gerekir.

Post post = new();

post.Title = "Roger Penrose Kimdir?";

post.BlogId = 3;

await context.Posts.AddAsync(post);

await context.SaveChangesAsync();

ÇOKA ÇOK İLİŞKİLERDE VERİ EKLEME

1. YÖNTEM DEFAULT CONVENTION KULLANILIYORSA;

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 ExampleDbContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("ConnectionStrings");

}

}

Book book = new();

book.BookName = "Beyaz Geceler";

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

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

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

await context.Books.AddAsync(book);

await context.SaveChangesAsync();

2.YÖNTEM EĞERKİ FLUENT API KULLANILYORSA;

class Book

{

public Book()

{

Authors = new HashSet<BookAuthor>();

}

public int Id { get; set; }

public string BookName { get; set; }

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

}

class BookAuthor

{

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<BookAuthor>();

}

public int Id { get; set; }

public string AuthorName { get; set; }

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

}

Bu durumda cross tablea erişimimiz olacaktır haliyle cross table üzerinden hem var olan bir entityle ilgili işlem yapabiliriz hem de bir entity oluşturabiliriz.

DbContext aşağıdaki gibi olacaktır;

3 idsine kitaba oluşturduğumuz yeni entityi eklerken, yeni bir kitap oluşturarak oluşturulan yeni entitye ekledik.

Author author = new()

{

AuthorName = "Alper Karataş",

Books = new HashSet<BookAuthor>()

{

new(){BookId=3},

new(){Book=new(){BookName="Uşak Nerededir?"}}

}

};

await context.AddAsync(author);

await context.SaveChangesAsync();

class ExampleDbContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=(LocalDb)\\HcSqlServer; Database=ExampleDatabase");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<BookAuthor>()

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

modelBuilder.Entity<BookAuthor>()

.HasOne(b => b.Book)

.WithMany(b => b.Authors)

.HasForeignKey(b => b.BookId);

modelBuilder.Entity<BookAuthor>()

.HasOne(a => a.Author)

.WithMany(a => a.Books)

.HasForeignKey(a => a.AuthorId);

}

}

İLİŞKİSEL TABLOLARDA VERİ EKLEME

Tüm işlemler aşağıdaki entity ve context sınıfları üzerinden gerçekleştirilmiştir.

Entityler:

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; }

}

Context:

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("ConnectionString");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Address>()

.HasOne(a => a.Person)

.WithOne(p => p.Address)

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

}

}

1-1 İlişkilerde Veri Güncelleme

Dependent veriyi Principal veri üzerinden güncelleme:

EFCore da principal veri üzerinden eriştiğimiz dependent veriyi güncellemek istiyorsak var olan dependent veri silinir ardından yeni veri eklenerek save edilir. Buradaki amaç ilişkiler üzerinden güncelleme işlemi yapmaktır.

Dependent verinin bağımlı olduğu principal veriyi değiştirme:

Person? person = await context.Persons

.Include(p => p.Address)

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

if (person!=null)

{

context.Remove(person.Address);

person.Address = new() { PersonAddress = "Yeni Adres" };

await context.SaveChangesAsync();

}

Öncelikle ilgili dependent veriyi silmemiz gerekmektedir. Veriyi sildikten sonra hangi principal veriye işlem yapılacaksa elde edilecek ardından principal veriye dependent veri olarak eklenecektir.

Silinen veri hala inmemoryde olduğu için güncelleme gerçekleşe cektir.

Address? address = await context.Addresses.FindAsync(3);

context.Remove(address);

await context.SaveChangesAsync();

Person person = await context.Persons.FindAsync(4);

person.Address = new() { PersonAddress = address.PersonAddress };

await context.SaveChangesAsync();

1-N İlişkilerde Veri Güncelleme

Dependent verileri Principal veri üzerinden güncelleme:

Tek bir dependent veri güncelleme:

Blog? blog = await context.Blogs.Include(b => b.Posts).FirstOrDefaultAsync(b => b.Id == 1);

if (blog!=null)

{

//idsi 2 olan postu güncelleme

Post? post= blog.Posts.FirstOrDefault(p => p.Id == 2);

post.Title = "Yeni Post Id 2";

}

await context.SaveChangesAsync();

Birden fazla dependent veri güncelleme:

Blog? blog = await context.Blogs.Include(b => b.Posts).FirstOrDefaultAsync(b => b.Id == 1);

if (blog!=null)

{

int a = 1;

foreach (var post in blog.Posts)

{

post.Title = $"Yeni Post{a}";

a++;

}

}

await context.SaveChangesAsync();

İlgili principal entity elde edilip sahip olduğu dependent entityler üzerinde değişiklik yapmak için aşağıdaki teknik kullanılır;

Blog? blog = await context.Blogs.Include(b => b.Posts).FirstOrDefaultAsync(b => b.Id == 1);

Post? silinecekPost = blog.Posts.FirstOrDefault(p=>p.Id == 2);

blog.Posts.Remove(silinecekPost);

blog.Posts.Add(new() { Title="Yeni Blog4"});

blog.Posts.Add(new() { Title="Yeni Blog5"});

await context.SaveChangesAsync();

Dependent verilerin bağımlı olduğu principal veriyi değiştirme:

Eğer ki dependent entity var olmayan bir principal entity ile güncelleme işlemine tabi tutulacaksa;

Önce dependent elde edilir ardından yeni principal eklenerek güncelleme yapılır.

Post? post4 = await context.Posts.FindAsync(4);

post4.Blog = new()

{

Name = "Blog 2"

};

await context.SaveChangesAsync();

Eğer ki dependent entity var olan bir principal entity ile güncelleme işlemi yapılacaksa;

Öncelikle principal ve dependent entity elde edilir dependent entitynin principalına güncellenecek olan principal entity verilerek güncellemeye tabi tutulur.

Post? post5 = await context.Posts.FindAsync(5);

Blog? blog2 = await context.Blogs.FindAsync(2);

post5.Blog = blog2;

await context.SaveChangesAsync();

N-N İLİŞKİLERDE VERİ GÜNCELLEME

1.Örnek

2.Örnek

//1 idli kitaba 3 idli yazarıda ekleme

Book? book = await context.Books.FindAsync(1);

Author? author = await context.Authors.FindAsync(3);

book.Authors.Add(author);

await context.SaveChangesAsync();

3.Örnek

//3 idsine sahip yazarın 1 idli kitap haricinde diğer kitaplarla ilişiği olmasın

Author? author = await context.Authors.Include(a => a.Books).FirstOrDefaultAsync(a => a.Id == 3);

foreach (var book in author.Books)

{

if (book.Id!=1)

{

author.Books.Remove(book);

}

}

await context.SaveChangesAsync();

// 2 idsine sahip olan kitabın yazarlarından idsi 2 olanı silip 4. Yazar adlı bir yazar ekle

Book? book = await context.Books.Include(b => b.Authors).FirstOrDefaultAsync(b => b.Id == 2);

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

book.Authors.Remove(silinecekAuthor);

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

await context.SaveChangesAsync();

İLİŞKİSEL TABLOLARDA VERİ SİLME

1-1 İlişkilerde Veri Silme

Yukarıda person principal adres ise dependent entitydir.

Person? person = await context.Persons

.Include(p => p.Address)

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

if(person!=null)

context.Addresses.Remove(person.Address);

await context.SaveChangesAsync();

1-N İlişkilerde Veri Silme

Blog? blog = await context.Blogs

.Include(b => b.Posts)

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

Post? post= blog.Posts.FirstOrDefault(p => p.Id == 2);

blog.Posts.Remove(post);

await context.SaveChangesAsync();

N-N İlişkilerde Veri Silme

Çoka çok ilişkilerde cross table ile bağlar kopartılır.

Author? author = await context.Authors

.Include(a => a.Books)

.FirstOrDefaultAsync(a => a.Id == 2);

Book? book=author.Books.FirstOrDefault(b => b.Id == 2);

author.Books.Remove(book);

await context.SaveChangesAsync();

Cascade Delete:

Principal tabledan herhangi bir veriyi silmeye çalıştığımızda, ilgili tableın dependent tablosunda nasıl bir davranış sergilemesi gerektiğini belirtmemiz için cascade delete yapılanmasının modellenmesi gerekmektedir. Modelleme Fluent API ile konfigure edilebilir.

Cascade Delete yapılanması OnDelete fonksiyonuna DeleteBehavior enumu üzerinden aşağıdaki yapılar çağıralak yapılır.

Cascade: Principal tablo silindiğinde ilişkili dependent verilerinde silinmesini sağlar.

Set Null: Principal tablo silindiğinde ilişkili dependent verilere null değeri atar.

Restrict: Principal tablo silinmeye çalışıldığında ilgili principal tablonun dependent verileri varsa, silinme işlemi engellenir.

Bu yapılanmayı EFCore default olarak Cascade olarak yapılandırmıştır. Yani herhangi özel bir konfigürasyon yapılmadığı takdirde yapılanma Cascade olacaktır.

Cascade:

modelBuilder.Entity<Post>()

.HasOne(b => b.Blog)

.WithMany(x => x.Posts)

.OnDelete(DeleteBehavior.Cascade);

Yukarıdaki durumda principal entity silindiğinde ilgili principal entitynin tüm dependent entityleri silinecektir.

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

context.Blogs.Remove(blog);

await context.SaveChangesAsync();

Principal entity olan blog silindiğinde tüm dependent entityler de silinecektir.

SetNull: Bu yapılanma 1-1 olan ilişkilerde yapılamaz çünkü id hem primary hem de foreign key olarak kullanılmaktadır.

İlişkisel tablolarda ilgili principal tablo silindiğinde dependent tablonun sahip olduğu foreign key kolonu nullable olmalıdır.

class Post

{

public int Id { get; set; }

public int? BlogId { get; set; }

public string Title { get; set; }

public Blog Blog { get; set; }

}

Bu işlemi yaptıktan sonra OnDelete içerisinde ilgili behavior verilip IsRequired fonksiyonu ile foreign key nullable olması için false değeri veriyoruz.

Artık ilgili principal entity silindiğinde dependent entitylerine null değeri atanacktır.

Blog? blog = await context.Blogs.FindAsync(2);

context.Blogs.Remove(blog);

await context.SaveChangesAsync();

modelBuilder.Entity<Post>()

.HasOne(b => b.Blog)

.WithMany(x => x.Posts)

.OnDelete(DeleteBehavior.SetNull)

.IsRequired(false);

Restrict: Bu yapılanmada dependent entityleri bulunan principal silinmeye çalışıldığında EFCore izin vermeyecek hata durumu ortaya çıkacaktır.

modelBuilder.Entity<Post>()

.HasOne(b => b.Blog)

.WithMany(x => x.Posts)

.OnDelete(DeleteBehavior.SetNull)

.IsRequired(false);

N-N ilişkililerde cascade delete yapılanması Cascade üzerinden yapılandırılır çünkü tablolar arası cross table vardır işlemler cross table üzerinden gerçekleştirilir yani ilgili veri cross table üzerinden yalın olarak silinir.

Backing Fields: Entity classlar içerisinde veritabanımızdaki kolonları propertyler ile değilde fieldlar ile temsil etmemizi sağlayan yöntemdir.

Shadow Property

Entity sınıflarında somut olarak tanımlanmayan ancak EFCore tarafında ilgili entity için var olduğu kabul edilen propertylerdir.

Tabloda gösterilmesine gerek duyulmayan entity instanceı üzerinden işlem yapılmayacak olan propertyler için kullanılabilir.

SP’lerin değerleri Change Tracker tarafından kontrol edilir.

class Blog

{

public int Id { get; set; }

public string Name { get; set; }

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

}

class Post

{

public int Id { get; set; }

public string Title { get; set; }

public DateTime LastUpdated { get; set; }

public Blog Blog { get; set; }

}

Yukardaki entitylerde foreign key olarak bir property tanımlanamamıştır ancak migration yaparken ilgili migrationunu incelersek EFCore bir foreign key kolonu oluşturduğunu göreceğiz bu kolonun oluşmasındaki etken Shadow Propertydir.

Shadow Property Oluşturma:

Bir shadow property oluşturmak için Fluent API kullanmalıyız. Blog entitye bir shadow property eklemek için;

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property<DateTime>("CreatedDate");

}

Shadow Property Değerini Okuma ve Güncelleme:

Change Tracker ile;



EF.Property üzerinden;

var blog = await context.Blogs.OrderBy(b => EF.Property<DateTime>(b, "CreatedDate")).ToListAsync();

EFCORE CONFIGURATIONS

Neden İhtiyacımız Olur?

Default davranışları bazı durumlarda özelleştirmeye ya da geçersiz kılmaya ihtiyacımız olur. Örneğin Entity classları içerisinde oluşturduğumuz bir propertynin ilgili veritabanında herhangi bir işlevi yoktur bu property programatik bir özellik olabilir. Bu gibi durumlarda özelleştirmelere ihtiyacımız olabilir.

OnModelCreating Metodu: Yapılandırma diyince akla ilk gelen metoddur. Virtual bir metoddur. Bu metod kullanılarak modellarda temel yapılandırmalar yapılır (Fluent API).

GetEntityTypes Metodu: Oluşturduğumuz entityleri programatik şekilde elde etmemizi sağlar.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

var entities = modelBuilder.Model.GetEntityTypes();

foreach (var entity in entities)

{

Console.WriteLine(entity.Name);

}

modelBuilder.Entity<Department>()

.Property(p => p.Name)

.HasColumnName("DepartmentName")

.HasColumnType("text")

.HasColumnOrder(7);

}

DataAnnotations | Fluent API

Aşağıdaki başlıkların solda olan DA ile sağda olanlar ise FAPI ile alakalıldır.

Table | ToTable:

Generate edilecek olan tablonun ismini belirlediğimiz yapılanmadıdır.

EFCore default olarak tabloların ismini oluşturulan DbSet propertynin adından alır.

Table Attribute ile:

[Table("Kisiler")]

class Person

{

//properties

}

ToTable API ile:

modelBuilder.Entity<Person>().ToTable("Kisiler");

Yukarıdaki işlemden sonra oluşturulacak olan tablonun adı Kisiler olacaktır.

Column | HasColumnName, HasColumnType, HasColumnOrder:

Veritabanındaki tabloların kolonlarını entity sınıfları içerisindeki propertyler temsil etmektedir. Bu durumda default olarak kolon isimleri property isimleriyken, kolon türleri propertynin türlerindedir.

Generate edilecek kolon isimlerine ve türlerine ve sırasına müdahale edebilmek için bu konfigürasyonlar kullanılır.

Column Attribute ile:

class Department

{

public int Id { get; set; }

[Column("DepartmentName", TypeName = "text", Order = 7)]

public string Name { get; set; }

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

}

Fluent API ile:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Department>()

.Property(p => p.Name)

.HasColumnName("DepartmentName")

.HasColumnType("text")

.HasColumnOrder(7);

}

ForeignKey | HasForeignKey

Foreign key ilişkisel tablolarda dependent entitynin, principal entitye karşı olan ilişkisini tutan kolondur.

Person-Department arası ilişki 1-n olsun. Bu durumda;

Foreign Key Attribute ile:

HasForeingKey Fluent API ile:

class Person

{

public int Id { get; set; }

[ForeignKey(nameof(Department))]

public int DId { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public Department Department { get; set; }

}

Fluent apida çalışırken tablolar arası ilişkinin öncelilkle belirlenmesi lazım.

modelBuilder.Entity<Person>()

.HasOne(p => p.Department)

.WithMany(d => d.Persons)

.HasForeignKey(p => p.DId);

NotMapped | Ignore

EFCore modellenen sınıf içerisindeki tüm propertyleri migrate sonucunda oluşturulacak olan tabloya kolon olarak ekler.

Bazı durumlarda modellenen entity içerisindeki propertynin kolon map edilmemesini isteyebiliriz. Sonuç olarak bu gibi durumlarda NotMapped ya da Ignore kullanılır.

NotMapped Attribute ile:

class Person

{

public int Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

[NotMapped]

public int KolonOlmasın { get; set; }

}

Ignore Api ile:

modelBuilder.Entity<Person>().Ignore(p=>p.KolonOlmasın);

Key | HasKey

EfCore da Id, ID, EntityId, EntityID isimli propertylere default olarak primary key constraint uygulanır.

Kendi irademizle istediğimiz isimdeki propertye primary key constraint uygulamak için Key attribute ya da Fluent Api ile HasKey kullanırız.

Key Attribute ile:

HasKey Api ile:

modelBuilder.Entity<Person>().HasKey(p => p.PrimaryKeyOlsun);

class Person

{

[Key]

public int PrimaryKeyOlsun { get; set; }

public string Name { get; set; }

}

Timestamp | IsRowVersion

Bir verinin bütünsel olarak değişikliğini takip etmemiz gerekebilir. Bu durumda verinin versiyonunu oluşturmamızı sağlayan konfigürasyondur.

Timestamp Attribute ile:

Fluent API ile:

modelBuilder.Entity<Person>()

.Property(p => p.RowVersion)

.IsRowVersion();

class Person

{

public int Id { get; set; }

public string Name { get; set; }

[Timestamp]

public byte[] RowVersion { get; set; }

}

Required | IsRequired

Bir kolonun null olup olmama durumunu konfigure etmemizi sağlar.

Required Attribute ile:

class Person

{

//nullable olma durumu ? operatörü ile yapılır

public int Id { get; set; }

[Required]

public string Name { get; set; }

}

IsRequired Api ile:

modelBuilder.Entity<Person>()

.Property(p => p.Name)

.IsRequired();

MaxLenght | HasMaxLenght

Bir kolonun max karakter sayısını konfigure etmemizi sağlar.

MaxLenght Attribute:

İlgili propertyde [MaxLenght(KarakterSayısı)] şeklinde tanımlanır

HasMaxLenght Api ile:

Precision | HasPrecision

modelBuilder.Entity<Person>()

.Property(p => p.Surname)

.HasMaxLength(30);

Küsüratlı değer tutan kolonlarda toplam karakter sayısı ve virgülden sonra kaç karakter tutulacağını konfigure etmemizi sağlayan yapılanmadıdır.

Precision Attribute ile:

[Precision(ToplamKarakterSayisi,VirgülSonrasıKarakterSayisi)]

HasPrecision Api ile:

modelBuilder.Entity<Person>()

.Property(p => p.Salary)

.HasPrecision(5, 3);

class Person

{

public int Id { get; set; }

public string Name { get; set; }

[Precision(5,3)]

public decimal Salary { get; set; }

}

Unicode | IsUnicode

İlgili kolonda Unicode kullanılacaksa bu konfigürasyon kullanılır.

Attribute ile:

class Person

{

public int Id{ get; set; }

public string Name { get; set; }

[Unicode]

public string Surname { get; set; }

}

Fluent Api ile:

Comment | HasComment  
İlgili kolonlara veritabanı seviyesinde comment eklemizi sağlar.

modelBuilder.Entity<Person>().Property(p => p.Surname).IsUnicode();

Attribute ile:

class Personn

{

public int Id{ get; set; }

public string Name { get; set; }

[Comment("Bu kolon X kolonudur")]

public int X { get; set; }

}

Fluent API ile:

modelBuilder.Entity<Personn>()

.Property(p => p.X)

.HasComment("Bu kolon X kolonudur");

ConcurrencyCheck |IsConcurrencyToken

Bir verinin bütünsel olarak değişikliğini takip etmemiz gerekebilir. Bu durumda verinin token oluşturmamızı sağlayan konfigürasyondur.

ConcurrencyCheck Attribute ile:

class Personn

{

public int Id{ get; set; }

public string Name { get; set; }

[ConcurrencyCheck]

public int ConcurrencyToken { get; set; }

}

IsConcurrencyToken Api ile:

modelBuilder.Entity<Personn>()

.Property(p => p.ConcurrencyToken)

.IsConcurrencyToken();

InverseProperty

İki entity arasında birden fazla ilişki durumunda navigation propertyleri ayarlamamızı sağlayan yapılanmadır.

FLUENT API

public class Flight

{

public int FlightID { get; set; }

public int DepartureAirportId { get; set; }

public int ArrivalAirportId { get; set; }

public string Name { get; set; }

public Airport DepartureAirport { get; set; }

public Airport ArrivalAirport { get; set; }

}

public class Airport

{

public int AirportID { get; set; }

public string Name { get; set; }

[InverseProperty(nameof(Flight.DepartureAirport))]

public virtual ICollection<Flight> DepartingFlights { get; set; }

[InverseProperty(nameof(Flight.ArrivalAirport))]

public virtual ICollection<Flight> ArrivingFlights { get; set; }

}

Composite Key

Bir tabloda birden fazla PK oluşturmamızı sağlayan yapılanmaya denir. HasKey API ile işlenir.

modelBuilder.Entity<Personn>().HasKey(p => new{ p.Id, p.Id2 });

HasDefaultSchema  
EFCore üzerinden inşa edilen tablolar default olarak dbo şeması olarak atanır. Eğer ki şema değiştirilmek istiyorak bu yapılanma kullanılır.  
  
HasDefaultValue  
Eğerki kolona bir değer atanmadığında default olarak değer atanması işlemini gerçekleştirir.

modelBuilder.HasDefaultSchema("DBOX");

modelBuilder.Entity<Personn>()

.Property(p => p.Name)

.HasDefaultValue("Person Name");

HasDefaultValueSql

Eğerki kolona bir değer atanmadığında default olarak değer atanması işlemini gerçekleştirir.

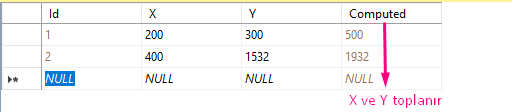
HasComputedColumnSql

modelBuilder.Entity<Personn>()

.Property(p => p.Name)

.HasDefaultValueSql("GETDATE()");

Veritabanında kolonlar arasında işlem yapmamızı sağlayan yapılanmadır.

Aşağıdaki tabloda X ve Y kolonlarını topladık.

HasData

modelBuilder.Entity<Example>()

.Property(e => e.Computed)

.HasComputedColumnSql("[X]+[Y]");

class Example

{

public int X { get; set; }

public int Y { get; set; }

public int Computed { get; set; }

}

Veritabanında inşa ederken bir yandan da hazır veriler barındırmak istiyorsak bu konfigürasyon kullanılır.

modelBuilder.Entity<Department>()

.HasData(new Department() { Id = 1, Name = "Abc" });

modelBuilder.Entity<Person>()

.HasData(

new Person {

Id = 1,

Name = "AAA",

DepartmentId = 1,

Surname = "BBB" },

new Person {

Id = 2,

Name = "BBB",

DepartmentId = 1,

Surname = "BBB" }

);