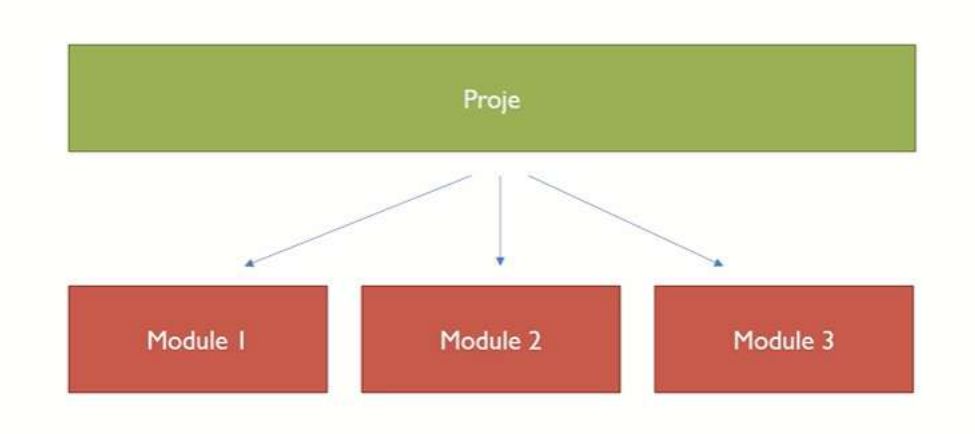
ASP.NET CORE-MVC

İçindekiler

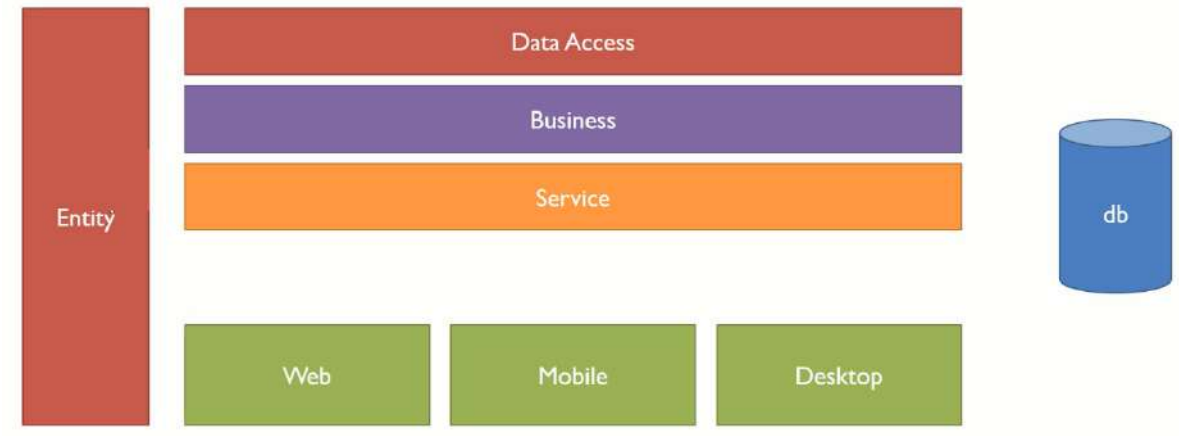
VERİTABANI ERİŞİMİ İÇİN MODÜLER YAPININ OLUŞTURULMASI

Modüler Yapıya Neden İhtiyaç Duyuyoruz?

Projemizin içinde yürüttüğümüz görevleri birbirinden ayrı ayrı görevlere bölüp bunları birbirinden bağımsız modüllere bölmemiz, ileriki geliştirme süreçlerinde bize kolaylık sağlayacaktır. Örneğin ilerleyen süreçte Veritabanı değişikliğine gidecek olursak ilgili veritabanıyla ilgili işlemleri bir modülde toplarsak, yeni geçtiğimiz veritabanında sadece ilgili modülü değiştiririz. Diğer modüllerimiz ondan bağımsız olduğu için yine çalışabilmelidir.

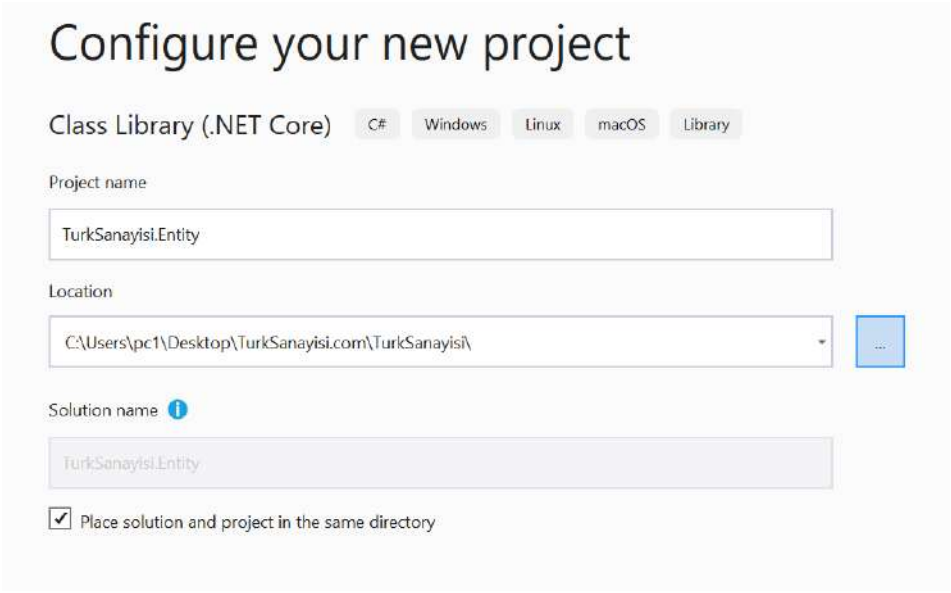


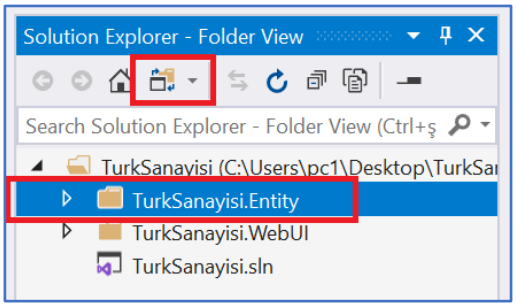
Kullandığımız veritabanı içindeki bilgilere üç farklı ortamdan erişebiliriz. Web, Mobil ve Masaüstü uygulamaları olabilir. Veritabanı iletişimi için bu üç uygulamanın herbiri içine ayrı ayrı kodları yazmamız gerekir. Oysa bu işlem için ortak bir veri erişim katmanı oluşturup burası üzerinden bağlantı sağlarsak aynı kodları üç kez yazmak durumunda kalmayız. Bu işlem için bir Data Access katmanına ihtiyaç vardır. İleride veritabanı değiştiği zaman ise Data Access yapıları değişecektir fakat bizim iş süreçlerinde kullandığımız tip tanımları ve bilgi alanları değişmeyecektir. Bu bilgilerin kaybolmaması için arada bir Business katmanı oluşturabiliriz. Bu katmanda veritabanı değiştiğinde değişmeyen bilgileri tutabiliriz. Kullandığımız arayüzlerden Web, Mobile, Desktop hepsi aynı platformda çalışan yapıda olmayabilir. Örneğin hepsi .net uyumlu olmayabilir. Bu durumda veritabanından gelen bilgileri hepsininde anlayacağı bir yapıda oluşturmamız gerekir. Bunun için arada bir Service katmanı oluşturmalıyız. Tüm işlemlerimizin sınıf yapılarını da Entity platformunda oluşturup, sınıf yapılarımızı buradan çekebiliriz. Böyle bir uygulama için modüler ve katmanlı bir erişim yapısına ihtiyaç vardır.



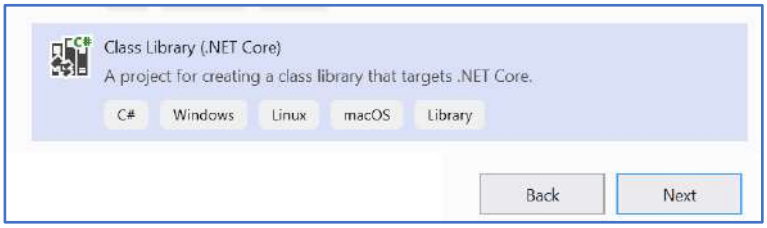
Entitiy Katmanın Eklenmesi (Sınıf Yapılarının olduğu katmanın Eklenmesi)

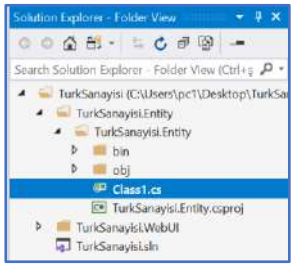
Not: Solution içinde ikinci bir proje oluşturmamız gerektiğinden Üst solution klasöründen itibaren proje tekrar yeniden oluşturuldu. Böylece Solution içinde iki tane proje oluşturulmuş olacak. Bunlar arayüzü temsil eden WebUI ve sınıf yapısını tutacağımız Entity projesi. Solution klasörü ise TS olacak.



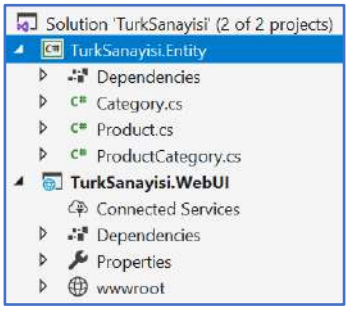


Bu klasör içinde oluşturacağımız proje tipi “Class Library (.Net Core)” tipinde olacak.





Entity katmanı içerisinde sınıf yapılarımızı oluştaracağız. Bu yapılar daha önceki oluşturduğumuz model yapıları içinde vardı. ProductMode, CategoryModel gibi. Bu yapıları Entity yapısı içinde oluşturacağız. İsimlerini değiştirelim. Kelime sonlarındaki Model ifadesini kaldıralım. Burada oluşturduğumuz 3.bir sınıf olan ProductCategory.cs sınıfı, ürünleri ve kategorileri birbirine bağlarken birden fazla kayıt karşılığı ile eşleştirebilmek için konulmuştur. Dosya içerikleri aşağıdaki şekildedir.



Category.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Entity
{
    class Category
    {
        public int CategoryId { get; set; }
        public string Name { get; set; }

        //Bir kategoride birden fazla ürünü tutmak
için
        public List<ProductCategory> ProductCategory {
get; set; }

    }
}
```

Product.cs

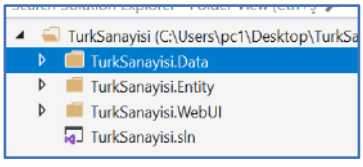
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Entity
{
    class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }
        public double? Price { get; set; }
        public string Description { get; set; }
        public string ImageUrl { get; set; }
        public bool IsApproved { get; set; }

        //Bir ürünü birden fazla kategoride tutmak
için
        public List<ProductCategory> ProductCategory {
get; set; }
    }
}
```

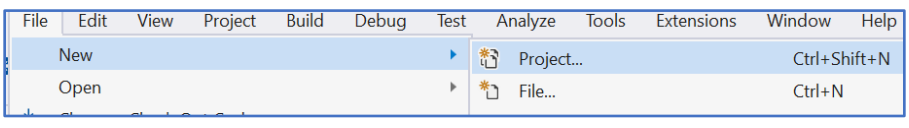
ProductCategory.cs

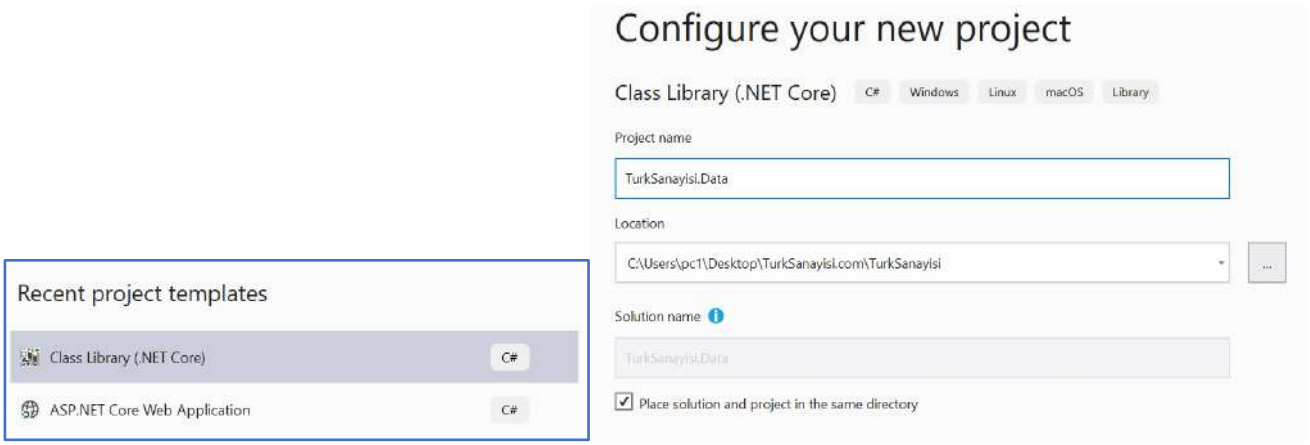
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Entity
{
    class ProductCategory
    {
        public int CategoryId { get; set; }
        public Category Category { get; set;
}
        public int ProductId { get; set; }
        public Product Product { get; set; }
    }
}
```

Data Katmanın Eklenmesi (Repository Katmanın Eklenmesi/Veritabanı Türüne bağlı sınıf yapılarının eklenmesi)

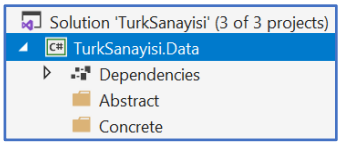
Entity katmanında olduğu gibi burada da yine bir tane Data klasörü oluşturalım ve içerisinde Class Library tipinde projemizi oluşturalım.







Data Projemiz içinde gelen hazır Class1.cs silelim.Ve iki tane "Abstract" ve "Concrete" isminde klasör ekleyelim.



Abstract Klasörü İçindeki İşlemler (Veritabanı İşlemlerinin olduğu sınıflar-Repository sınfları)

Abstract içerisine bir tane Interface class yapısı ekleyelim. İsmini de `IproductRepository` verelim. Sınıf adının başına public ifadesini ekleyerek başka klasörlerden erişime açalım. Aşağıdaki şekilde bu Interface içerisine Veritabanı ile ilgili işlemleri yapacağımız temel uygulamaları ekleyelim. Örneğin GetById metoduna Id bilgisini gönderelim o da bize geri bir ürün bilgisi göndersin. Bu ifade şu şekilde yazılmış oldu. Product GetByeId(int Id);

IProductRepository.cs

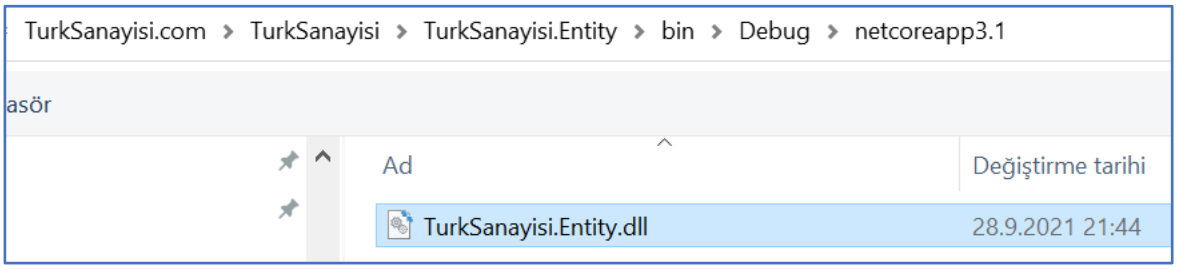
```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IProductRepository
    {
        Product GetById(int Id);
        List<Product> GetAll();
        void Create(Product Entity);
        void Update(Product Entity);
        void Delete(int Id);

    }
}
```

Burada Product sınıfını program görmeyecektir. Bu sınıf Entity projesi içindeydi. Data projesi içinden Entity projesindeki sınıfı kullanabilmek için birbirine tanıtmamız gerekir. Bunun için aşağıdaki adımları takip edin.

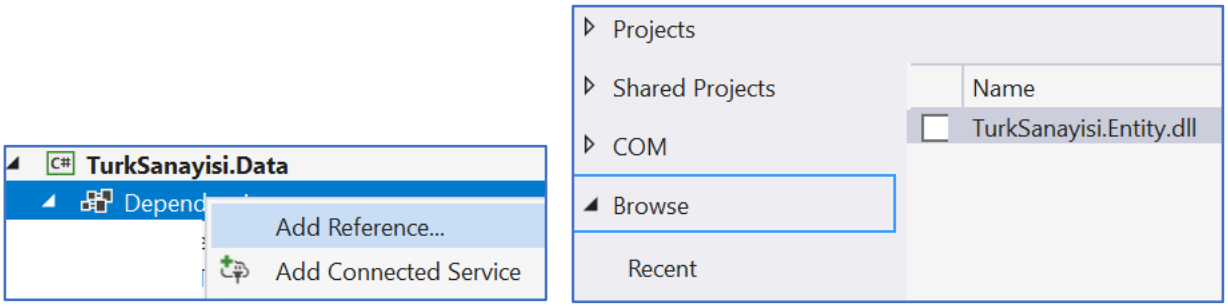
Projeye Reference Ekleme İşlemi

Not (Önemli): Data projesi içinden Entity projesini çağırmak için Reference ekleme için şu adımları takip edin.

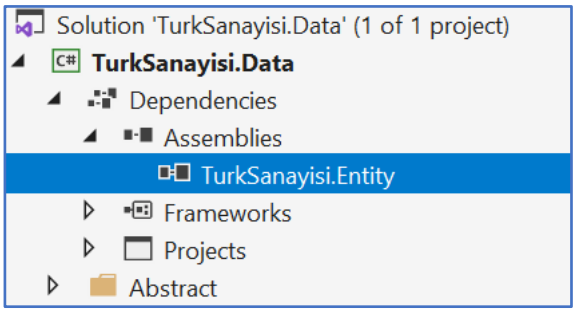
a) Önce Entity projesini "Build" işlemi ile derleyin. Böylece aşağıdaki yol içinde .dll dosyalarının oluştuğundan emin olun



b) Sonra Data projesi içinde iken Depencies üzerinde sağ tuşa tıklayıp Reference eklemeyi seçin. Sonra açılan pencereden Browserı kullanarak yukarıdaki yolun içindeki .dll dosyasını işaretleyin ve ok tıklayın.



c) Bu işlemi yaptığımızda Data projesi içerine Entity projesi eklenmiş olur.



d) Aynı zamanda Data projesinin dosyası içerisine aşağıdaki şekilde Entity projesinin yolunu eklemiş olduğunu görelim.

Şimdi benzer şekilde ICategoryRepository.cs sınıfını da Abstract klasörü içinde oluşturalım. Kodlar şu şekilde olacaktır.

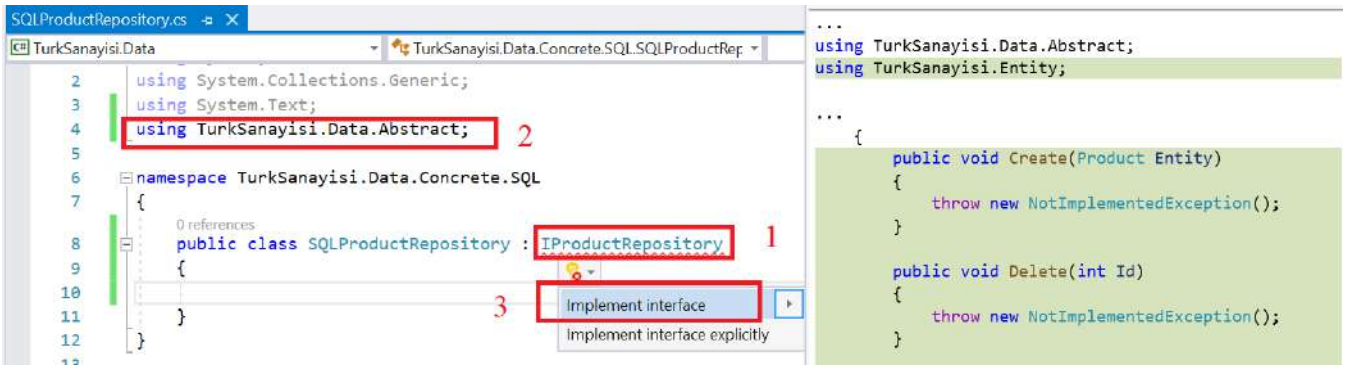
ICategoryRepository.cs

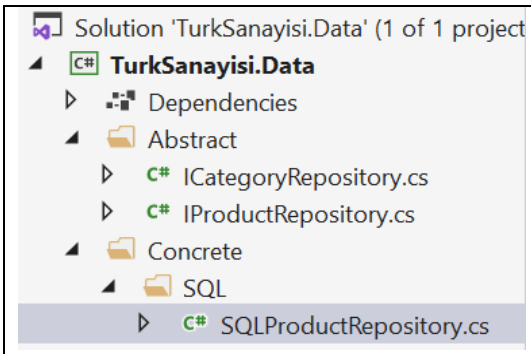
```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface ICategoryRepository
    {
        Category GetById(int id);
        List<Category> GetAll();
        void Create(Category entity);
        void Update(Category entity);
        void Delete(int id);
    }
}
```

Concrete Klasörü İçindeki İşlemler (Veritabanı tipine göre içi dolu sınıfların oluşturulması)

Burada Data.Abstract klasörü içinde oluşturduğumuz IproductRepository bir sanal sınıf olmuş oldu. Bu sanal sınıfın dolu hali Concrete klasörü içinde oluşturulacak. Bunun için Concrete içerisine SQLServer veritabanı işlemlerinde kullanmak üzere bir SQL klasörü oluşturalım. Bu klasör içindede SQLProductRepository sınıfını oluşturalım. Bu class içindeki yapı Absract içindeki IproductRepository den alınacak. Böylece ileride SQL veritabanı yerine başka bir veritabanına geçilirse Repository yapısı direk IproductRepository sınıf yapısı içinden alınacak. Böylece program modüler ve katmanlı bir yapı kazandığı için ileride olabilecek değişikliklere kolay uyum sağlanacak.

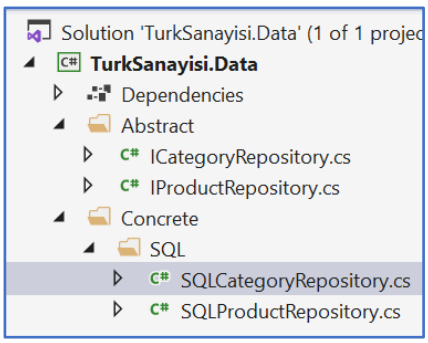
Bu sınıfı iç yapısı IproductRepository den geleceği için bu sınıf implement etmeliyiz (1 nolu madde). Ardından sınıfın bulunduğu klasörü yukarıya eklemeliyiz (2 nolu madde). Ardından oradaki yapının kullanılacak şekilde gelmesi için 3 nolu işaretteki seçimi yapmalıyız. Böylece sınıfın içi aşağıdaki şekilde otomatik olarak dolacaktır.






```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.SQL
{
    public class SQLProductRepository : IProductRepository
    {
        public void Create(Product Entity)
        {
            throw new NotImplementedException();
        }

        public void Delete(int Id)
        {
            throw new NotImplementedException();
        }

        public List<Product> GetAll()
        {
            throw new NotImplementedException();
        }

        public Product GetById(int Id)
        {
            throw new NotImplementedException();
        }

        public void Update(Product Entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

Benzer şekilde kategori için de SQLCategoryRepository i oluşturalım.

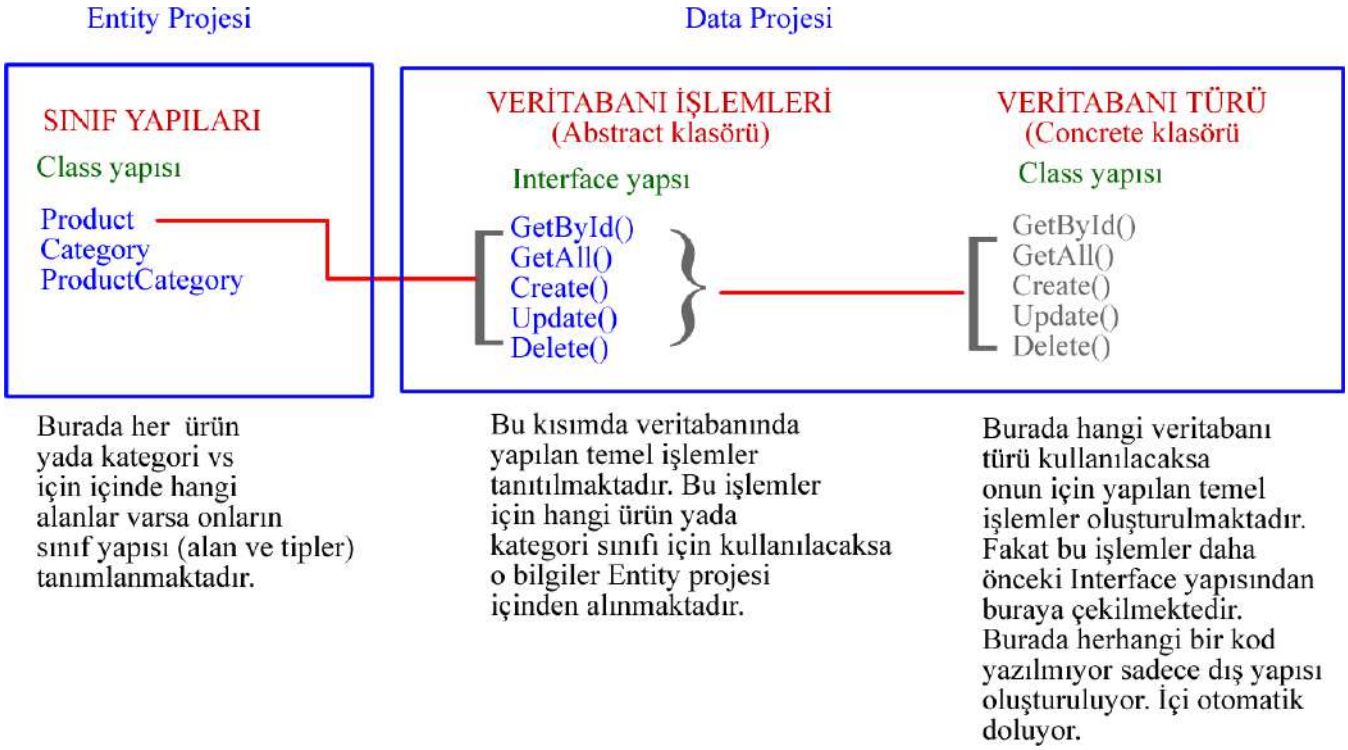


SQLCategoryRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Data.Abstract;
```

```
using TS.Entity;

namespace TS.Data.Concrete.SQL
{
    public class SQLCategoryRepository : ICategoryRepository
    {
        public void Create(Category entity)
        {
            throw new NotImplementedException();
        }

        public void Delete(int id)
        {
            throw new NotImplementedException();
        }

        public List<Category> GetAll()
        {
            throw new NotImplementedException();
        }

        public Category GetById(int id)
        {
            throw new NotImplementedException();
        }

        public void Update(Category entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

Özet: Buraya kadar olan işlemlerin özet şeması şu şekildedir.



Generic Interface Oluşturulması (Interface yapısındaki (Abstract içindeki)dosyaların yapısını tek bir dosyadan çekme)

Data projesi içindeki Abstract klasörünün içindeki dosyaları incelediğimizde IProductRepository, ICategoryRepository içindeki tüm yapıların birbirinin benzeri olduğunu görüyoruz. Bunlardan başka IOrderRepository de oluştursak aynı yapıyı onunda içinde kullanmış olabiliriz. Dolayısı ile bütün bu Repository

yapılarının (Veritabanı işlemlerinin olduğu dosyaları) olduğu dosyaları tek bir dosya ile oluşturabiliriz. Bunun için ortak bir isim olarak IRepository.cs adını kullanalım. Bu dosya içinde dışarıdan gönderdiğimiz tip için <T> ifadesini kullanalım. Yada <TEntity> olabilir. Yani dışarıdan hangi tip entity gelirse (Product, Category, Order vs) onu metod içinde bu isimle temsil edeceğiz.

IRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TS.Data.Abstract
{
    public interface IRepository<TEntity>
    {
        TEntity GetById(int Id);
        List<TEntity> GetAll();
        void Create(TEntity Entity);
        void Update(TEntity Entity);
        void Delete(int Id);
    }
}
```

Artık bir şablon yapı olarak görebileceğimiz Repository yapısını oluşturduk. Artık daha önce oluşturduğumuz Repositorylerde (IProductRepository, ICategoryRepository) bu yapıyı kullanalım.

IProductRepository nin için aşağıdaki gibi boşaltalım. İçerisindeki bu yapıyı IRepository şablonundan alacağı için bu satırlara gerek kalmadı.

IProductRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IProductRepository: IRepository<Product>
    {

    }
}
```

Gördüğümüz gibi bu yapı içinde herhangi bir metod bırakmadık. VT ile ilgili metodlar IRepository içinden gelecektir. Ordaki yapıyı kullanırkende Tentity yerine Product entitisini ekleyecektir.

Benzer şekilde ICategoryRepository'sini de oluşturalım.

ICategoryRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface ICategoryRepository : IRepository<Category>
    {

    }
}
```

Order Entity Yapısının Oluşturulması (Yeni Bir Entity Yapısı Oluşturma)

Şimdi bu modüler yapının bir avantajını kullanalım. Örneğin projemizin ilerleyen aşamalarında şiparişler için bir Order bilgisine ihtiyaç olsun. Öncelikle bu bilgi için hangi bilgilerin tutulacağı sınıf yapısını Entity projesi içinde oluşturalım. Ardından Veritabanı işlemlerine geçtiğimizde (Repository yapısına ihtiyaç duyduğumuzda) hızlı bir şekilde yapıyı oluşturabiliriz. Dosyalarımız aşağıdaki şekilde olmuş olur.

TS.Entity>Order.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TS.Entity
{
    public class Order
    {
        public int OrderId { get; set; }
    }
}
```

TS.Data>Abstract> IOrderRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IOrderRepository :
IRepository<Order>
    {

    }
}
```

Not: Burada Order.cs dosyasını Entity projesi içerisine sonradan ekledik. .dll dosyasının içinde bu class ın yer alabilmesi için F5 ile bir defa çalıştırılmalı (Build yapılmalı). Bu işlem yapılmadan IOrderRepository içine using TS.Entity; namespaci ni ne kadar yukarıya eklesek de Order entitisini görmeyecektir.

Dikkat edilirse burada IOrderRepository yi oluşturduk ve içerisine hiçbir Veritabanı işlemleri için metod yazmadık. Hazır bir şekilde tüm metodlar IRepository dosyasından gelmektedir. Ordaki TEntity yerine Order nesnesi konularak aynen çalışmış olacaktır.

Standart Repository (VT işlemleri) Metodlarının Yanında Ekstra Metod Oluşturma

Şu ana kadar Repository içinde hep aynı metodları kullandık. Bunlar VT için standart uygulamalardır (GetByID, GetAll, Create, Update, Delete). Örneğin Product işlemleri yapılırken Popular ürünleri getirmek isteyebiliriz. Bu durumda tüm işlemleri (metodları) IRepository içinden çekiyorsak bu eksra işlemi nasıl ekleyeceğiz. Bunu da IProductRepository içine ayrıca eklersek diğerlerinin üzerine (standart metodların üzerine) bunu da eklemiş olur. Kodlar şu şekilde olacaktır.

IProductRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IProductRepository: IRepository<Product>
    {
        List<Product> GetPopularProducts();
    }
}
```

Aynı yöntemi kullanarak kategori içinede ekstra bir metod ekleyelim. Buda bize populer kategorileri getirsin. Metodlar geriye Category içeren bir liste göndereceğinden List<Category> metodun başına eklenmektedir.

ICategoryRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Entity;
```

```
namespace TS.Data.Abstract
{
    public interface ICategoryRepository : IRepository<Category>
    {
        List<Category> GetPopularCategories();
    }
}
```

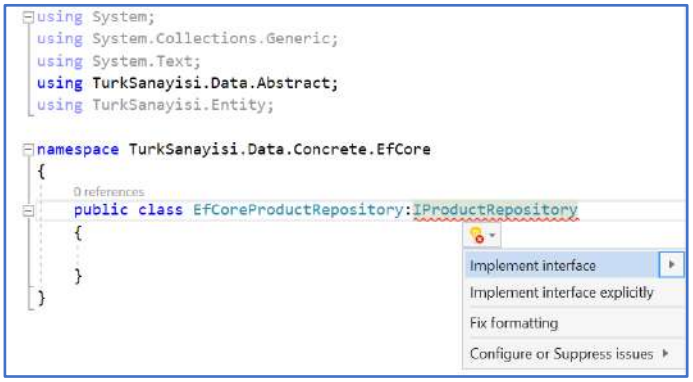
EntityFrameworkCore'un (Veritabanı işlemlerini yapacak kütüphane) Proje içerisine eklenmesi

Yukarıdaki yaptğımız işlemlerde Concrete klasörü içinde SQL veritabanıyla ilgili işlemleri yapacak sınıfları koymuştuk. Bu sınıfları bir veritabanı üzerinde denemedik ama bu kısımda bir değişikliğe gidelim.

Concrete klasörü içinde Veritabanı türüne göre sınıfları oluşturmak yerine tüm veritabanları üzerinde işlem yapabilen veri erişim teknolojisi kullanalım. Bunlardan bir tanesi **EntityFrameworkCore** kütüphanesidir. Bundan başka "dapper" yada "Adonet" kütüphaneleri de olabilirdi. EfCore için aşağıda yapacağımız işlemleri diğer AdoNet gibi kütüphaneleri kullanacak olursak benzer şekilde yapabiliriz.

EntityFramework kütüphanesini projeye eklemeden önce alt yapısını bir kuralım. Bu amaçla önce Concrete>EfCore isimli bir klasör oluşturalım. Daha önceden orada oluşturduğumuz SQL klasörünü ve içindeki dosyaları kaldıralım.

EfCore klasörü içinde EfCoreProductRepository isminde bir class oluşturalım. Bu class dolu yapısı IProductRepository sınıfını kullanacaktır. Bu sınıfı implement edelim. Bu işlem için aşağıda gösterildiği şekilde İlgili başlık üzerine gelip "Show Potansiels" yazına tıklayıp "Implement Interface" ifadesini seçelim.



Bu seçim işleminden sonra class ın içerisi aşağıdaki şekilde dolacaktır. Yani Veritabanı üzerinde gerçekleştirilecek standart tüm medotlar gelmiş olmaktadır. Dikkat edilirse IproductRepository içerisine eksradan GetPopularProducts metodunu da eklemiştik. Bu metodda diğerleri ile birlikte gelmiş olmaktadır.

EfCoreProductRepository.cs

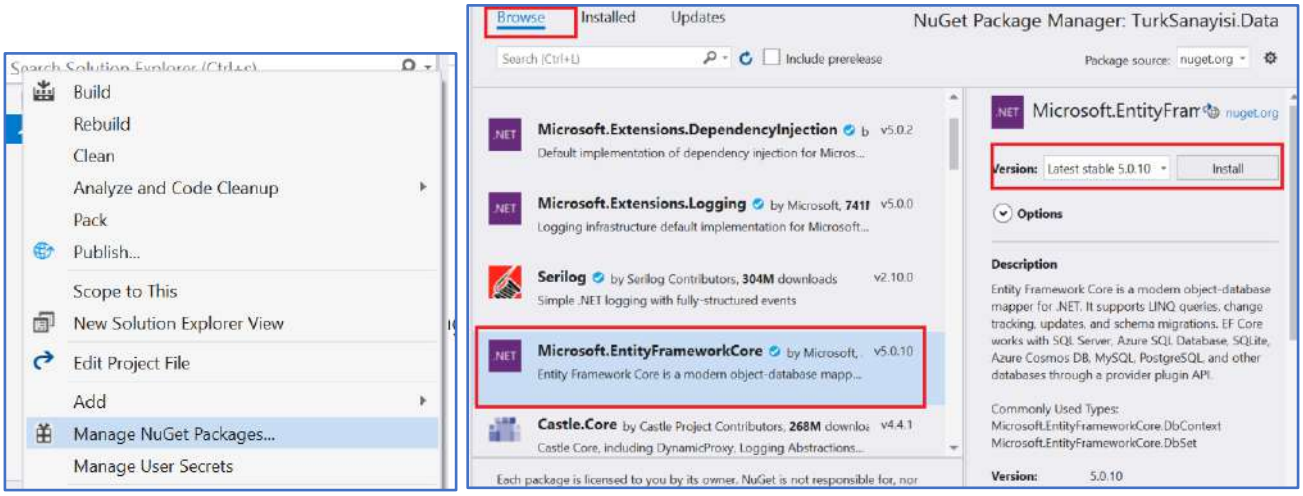
```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreProductRepository : IProductRepository
    {
        public void Create(Product Entity)
        {
            throw new NotImplementedException();
        }
        public void Delete(int Id)
        {
            throw new NotImplementedException();
        }
```

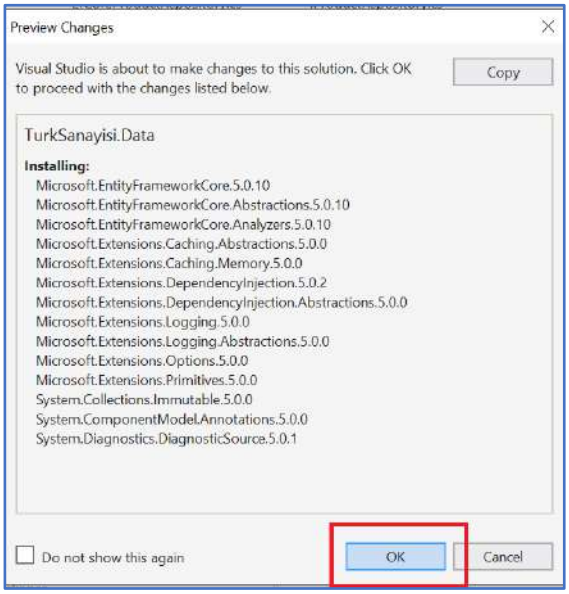
```
        public List<Product> GetAll()
        {
            throw new NotImplementedException();
        }
        public Product GetById(int Id)
        {
            throw new NotImplementedException();
        }
        public List<Product> GetPopularProducts()
        {
            throw new NotImplementedException();
        }
        public void Update(Product Entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

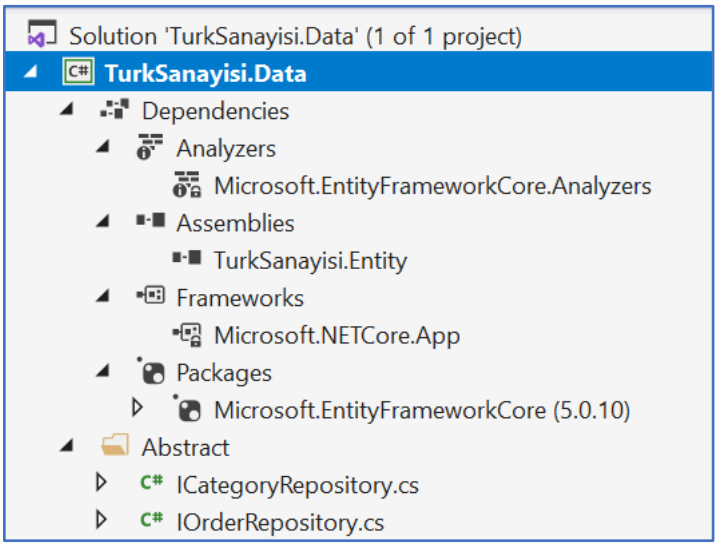
Ef (EntityFramework) için alt yapıyı hazırladık. Artık bu kütüphaneyi Data projemizin içine ekleyelim (Veritabanı işlemleri için başkalarının yazdığı hazır .dll dosyalarını kullanacağız).

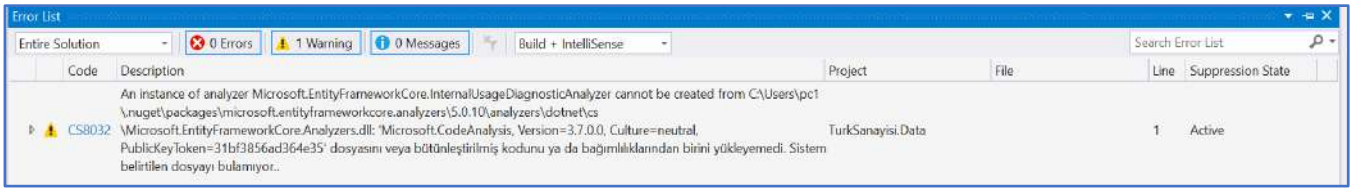
EfCore Kütüphanesinin Yüklenmesi

Yada aşağıdaki şekilde kurulabilir (Anlatım videosu: https://www.youtube.com/watch?v=2sDVVwojuvE). Proje üzerine sağ tıklayıp "Manage NuGet Packeges" işaretlenir. Ardından "Browse" tıklanır ve aşağıdaki seçimler yapılarak kurulum başlatılır.

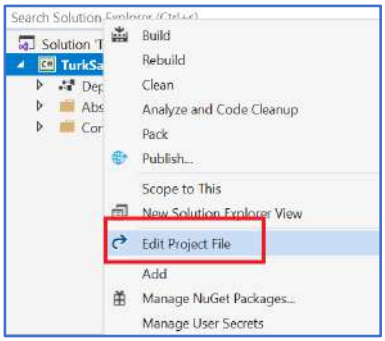


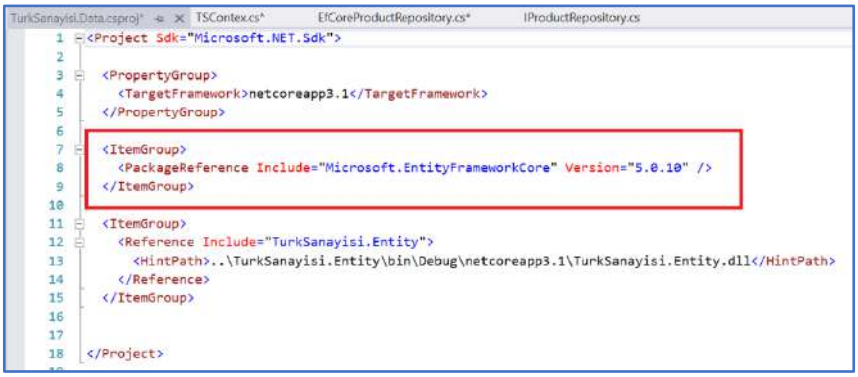






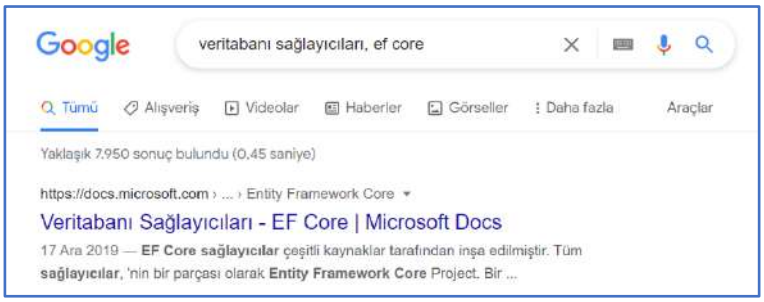
Entity Framework kurulduktan sonra csproj dosyası içine yolunu eklediğini görelim.





EfCore-için SqLite Veritabanı Kütüphanesinin Yüklenmesi

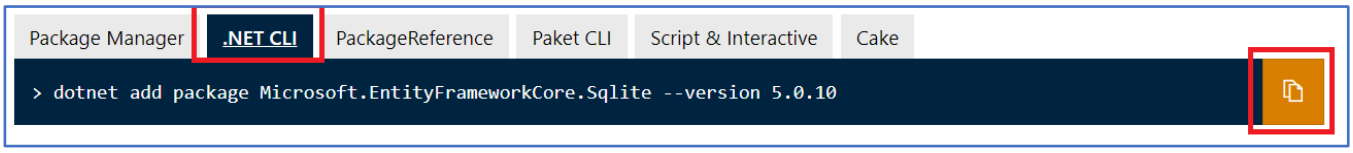
EfCore kütüphanesini SqLite veritabanı ile kullanabilmek için onunla alakalı kütüphaneyi de yükleyelim. Google şu ifadeyi yazalım (veritabanı sağlayıcıları, ef core).



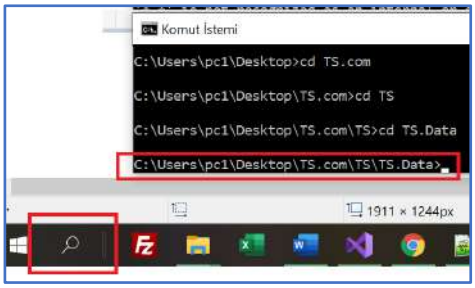
Açılan sayfada Sqlite veritabanı için olan kütüphaneyi indirelim. Bu veritabanı burada deneme amaçlı kullanacağız. Eğer SqlServer kullansaydık onun için olanı üstündeki linkten indirebilirdik. Bunu ileride kullanacağız.

NuGet Paket	Desteklenen veritabanı altyapıları	Bakımcı /Satıcı	Notlar / Gereksinimler
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2012 ve sonra	EF Core Project (Microsoft)	
Microsoft.EntityFrameworkCore.Sqlite	SQLite 3.7 ve sonra	EF Core Project (Microsoft)	

Açılan sayfada kütüphaneyi yükleyeceğimiz yolu kopyalayalım.

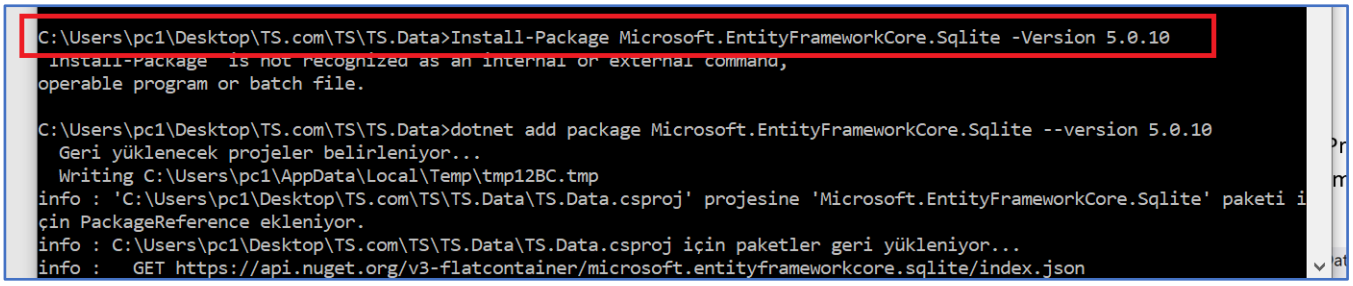


Yükleme linkini aldıktan sonra Windows komut satırına cmd yazıp Komut istem ekranını açalım.

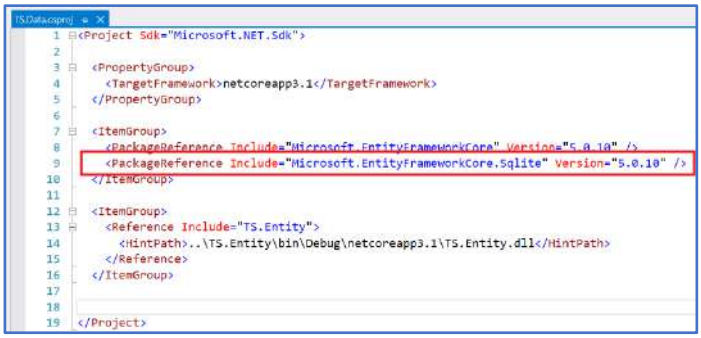


Siteden aldığımız yolu komut satırında ilgili Data projesinin içine konumlandıktan sonra yapıştıralım ve entera tıklayalım.

dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 5.0.10



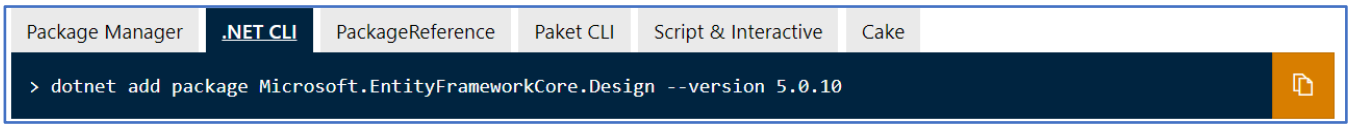
Böylece SqLite veritabanını kullanabilmek için dll leri kurmuş oluyoruz. Henüz SQL veritabanına geçmedik. Data projesinin csproj dosyası içinde ilgili kütüphanenin dahil edilmesine ait kodlar konulmuş durumda.



EfCore için Design Paketinin Yüklenmesi

Bu işlem için yukarıda SqLite paketinin yüklendiği şekilde Data projesinin içerisine komut satırından gidelim. Paketin yükleme adresini şu adresten alalım.

(https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Design/)



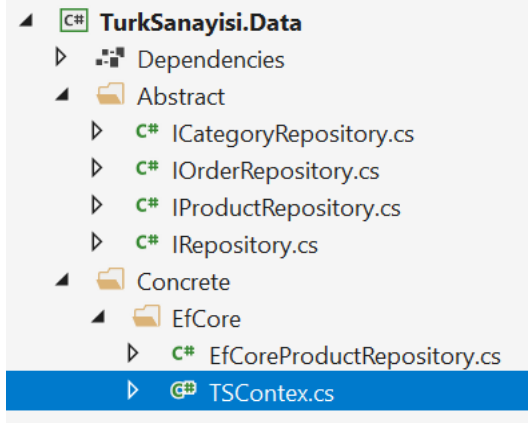

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet add package Microsoft.EntityFrameworkCore.Design --version 5.0.10
  Geri yüklenecek projeler belirleniyor...
  Writing C:\Users\pc1\AppData\Local\Temp\tmp7F6B.tmp
info : 'C:\Users\pc1\Desktop\TS.com\TS\TS.Data\TS.Data.csproj' projesine 'Microsoft.EntityFrameworkCore.Design' paketi i
çin PackageReference ekleniyor.
info : C:\Users\pc1\Desktop\TS.com\TS\TS.Data\TS.Data.csproj için paketler geri yükleniyor...
info :   GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json 529 ms
```

Bu işlemden sonra Data projesinin csproj dosyası içinde paketin yüklendiğini görüyoruz.

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="5.0.10" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="5.0.10">
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</I
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="5.0.10" />
</ItemGroup>
```

Şu anda EfCore Data projemizde kullanabiliriz.

EfCore klasörü içerisine ayrıca birde Context sınıfı oluşturalım. Adıda TSContext olsun. Bu sınıf EfCore DbContext sınıfından türetildi. Ürün ve Kategoriler için veritabanı işlemlerini içeren DbSet ler tanımlandı. Ayrıca en altta daha sonra proje içerisinde oluşturulacak olan TSDb veritabanın yolu tanımlandı.



TSContex.cs

```
using Microsoft.EntityFrameworkCore;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class TSContext : DbContext
    {
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }

        protected override void
OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=TSDb");
        }

    }
}
```

Şimdi bu veritabanı işlemlerini EfCoreProductRepository.cs içinden çağıralım. Örneğin aşağıda sadece ürün eklemek için kullandığımız Create metodu veritabanına ürün kaydederken kullanılacak ve kodları aşağıdaki şekildedir. Benzer şekilde EfCoreCategoryRepository.cs dosyasını da oluşturalım. Toptan kodları verelim.

EfCoreProductRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Data.Abstract;
using TS.Entity;
```

EfCoreCategoryRepository.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using TS.Data.Abstract;
using TS.Entity;
```

```
namespace TS.Data.Concrete.EfCore
{
    public class EfCoreProductRepository :
IProductRepository
    {
        private TSContext db = new TSContext();

        public void Create(Product Entity)
        {
            db.Products.Add(Entity);
            db.SaveChanges();
        }

        public void Delete(Product Entity)
        {
            throw new NotImplementedException();
        }

        public List<Product> GetAll()
        {
            throw new NotImplementedException();
        }

        public Product GetById(int Id)
        {
            throw new NotImplementedException();
        }

        public List<Product> GetPopularProducts()
        {
            throw new NotImplementedException();
        }

        public void Update(Product Entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
namespace TS.Data.Concrete.EfCore
{
    public class EfCoreCategoryRepository :
ICategoryRepository
    {
        private TSContext db = new TSContext();
        public void Create(Category Entity)
        {
            db.Categories.Add(Entity);
            db.SaveChanges();
        }

        public void Delete(Category Entity)
        {
            throw new NotImplementedException();
        }

        public List<Category> GetAll()
        {
            throw new NotImplementedException();
        }

        public Category GetById(int Id)
        {
            throw new NotImplementedException();
        }

        public List<Category> GetPopularCategories()
        {
            throw new NotImplementedException();
        }

        public void Update(Category Entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

Gördüğümüz gibi burada db sınıfını kullanarak ister ürünlere (Products) ister kategorilere (Categories) Add() metodunu kullanarak hangi entity geliyorsa (product yada category) onu kaydediyoruz. Uygulamalar aynı sadece Entity (produc, category vs) nin tipi değişiyor. Yapılan işlemler aynı. Dolayısı ile herbiri için ayrı sınıf oluşturmak yerine tek bir generic bir sınıf oluşturursak onunla hepsini (bütün entitileri=product, category, order vs) kaydedebilir yada diğer ve veritabanı işlemlerini yapabiliriz.

EfCore Generic Repository'nin oluşturulması (Benzer EfCore sınıflarını tek bir sınıftan yönetme)

EfCore klasörümüz Create klasörü içerisindeydi. Yani Abstract klasörü içindeki sınıflarımızın dolu haliydi. Veritabanı ile en yakın işlemlerin yapıldığı sınıflar idi. Bu EfCore klasörü içindeki dolu sınıflarımız olan EfCoreProductRepository, EfCoreCategoryRepository yada benzer şekilde "Order" gibi başka entitiler içinde oluşturmuş olabiliriz, bunların hepsinin içinde benzer veritabanı işlemleri vardı. İşte ortak bu işlemleri generic bir sınıf içine alalım ve herbiri için tekrar tekrar oluşturmak durumunda kalmayalım.

Bunun için EfCore içinde EfCoreGenericRepository isminde bir sınıf oluşturalım ve içini aşağıdaki şekilde dolduralım. Bu sınıf yukarıdaki kodlarda olduğu gibi bir TSContext isminde oluşturduğumuz sınıfa (İçerisinde DbSet ler bulunuyordu) kullanıyordu. Ama biz burada Generic (esnek her yer kullanabileceğimiz bir yapı) yaptığımızdan sınıf içinde kullanmak üzere bir TContext isminde hepsi için kullanılabilecek bir Context tanımlayalım. Bu sınıf içinde aynı zamanda Abstract klasörü içinde generic bir Interface oluştururken kullandığımız TEntity ismindeki parametreyide dışarıdan alalım. EfCoreGenericRepository ismiyle tamam oluşturduğumuz sınıf Abstract klasörü içinde IRepository içindeki metodları Implement edecek. Dolayısı ile bu sınıfın adını sınıf adının sonuna (:)

noktadan sonra eklemeliyiz. (Not: metodları imlement etmek için IRepository<TEntity> yazısının üzerine sağ tuşa tıklayıp "Implement Repository" seçilirse, o sınıfın içindeki tüm metodlar aşağıdaki şekilde gelecektir.). Sınıfa gönderilecek TEntity lerin hangi tipte olması gerektiğini tanımın altında veriyoruz. Mecbur değil ama verilmesi gönderilecek elemanın tipinin ne olması gerektiğini anlamak açısından önemlidir. Buradan anladığımız TEntity ve TContext girişleri class tipinde olacak ve bunlardan TContext, DbContext ten türetilebilir newlenebilir (çoğaltılabilir) bir class olacak demektir. Bu kısıma ait kodlar;

```
class EfCoreGenericRepository<TEntity, TContext> : IRepository<TEntity>
     where TEntity:class
     where TContext:DbContext,new()
```

Buradan anladığımız Database işlemler aktarılırken TSContext.cs içindeki DbContext üzerinden veritabanına gönderilecek. Kullanıcı ise DbContext sınıfını bize TSContext adı ile gönderecek.

Burada TContext, entityframeworks yapısını içeren ve o platforma ait DbContext ten türetilen bir sınıf olmuş oluyor. Burada metod içinde ondan bir nesne türetiyoruz (context nesnesi). Bu nesne içerisinde hangi entitiyi kullanacak ise (product, category, order vs gibi) onu parametre olarak ayarlıyoruz. Bunları temsil eden ise TEntity dir. TEntity içerisinde Product, Category yada Order geliyor olabilir. Bu entity üzerinde yapılacak işlem ise Add() metodu ile gösterilmiş oluyor. Bu kısıma ait kodlar;

```
public void Create(TEntity entity)
{
    using(var context=new TContext())
    {
        context.Set<TEntity>().Add(entity);
        context.SaveChanges();
    }
}
```

Aynı şekilde Silme işlemi için olan metodu yazalım.

```
public void Delete(TEntity entity)
{
    using (var context = new TContext())
    {
        context.Set<TEntity>().Remove(entity);
        context.SaveChanges();
    }
}
```

Listeleme metodunu da yazalım. toList() metodun için `using System.Linq;` namespace ni eklemeliyiz. Ayrıca bu metod geriye liste gönderdiğinden return ifadesini kullanmalıyız.

```
using System.Linq;

public List<TEntity> GetAll()
{
    using (var context = new TContext())
    {
        return context.Set<TEntity>().ToList();
    }
}
```

Id ye göre arama bulma metodu da aynı şekilde olacaktır.

```
public TEntity GetById(int Id)
{
    using (var context = new TContext())
    {
        return context.Set<TEntity>().Find(Id);
    }
}
```

Güncelleme işlemi için olan metodumuz aşağıdaki şekilde olur. Entry metodu kullanılır. Ve bunun hangi alanın güncelleneceği ise State işlemi ile belirtilir.

```
public void Update(TEntity entity)
{
    using (var context = new TContext())
```

```
            {
                context.Entry(entity).State = EntityState.Modified;
                context.SaveChanges();
            }
        }
```

Tüm kodları gösterirsek aşağıdaki şekilde olacaktır.

EfCoreGenericRepository.cs

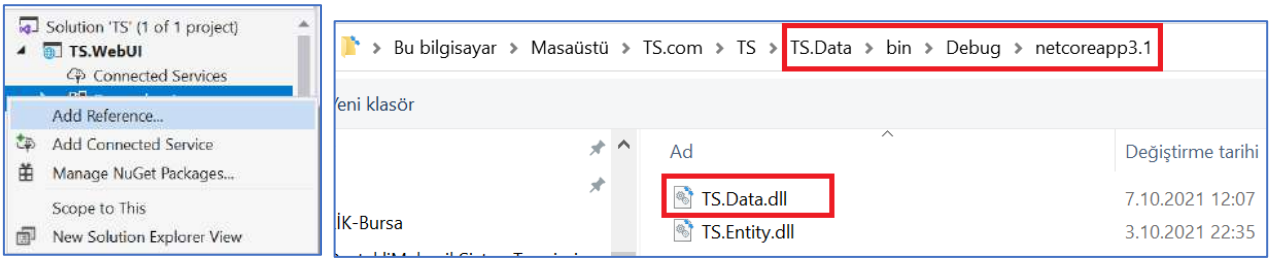
```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TS.Data.Abstract;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreGenericRepository<TEntity, TContext> : IRepository<TEntity>
        where TEntity:class
        where TContext:DbContext,new()
    {
        public void Create(TEntity entity)
        {
            using(var context=new TContext())
            {
                context.Set<TEntity>().Add(entity);
                context.SaveChanges();
            }
        }

        public void Delete(TEntity entity)
        {
            using (var context = new TContext())
            {
                context.Set<TEntity>().Remove(entity);
                context.SaveChanges();
            }
        }

        public List<TEntity> GetAll()
        {
            using (var context = new TContext())
            {
                return context.Set<TEntity>().ToList();
            }
        }

        public TEntity GetById(int Id)
        {
            using (var context = new TContext())
            {
                return context.Set<TEntity>().Find(Id);
            }
        }

        public void Update(TEntity entity)
        {
            using (var context = new TContext())
            {
                context.Entry(entity).State = EntityState.Modified;
                context.SaveChanges();
            }
        }
    }
}
```

Madem generic olarak veritabanı işlemlerini yapacak sınıfımızı oluşturduk. Bu sınıfı EfCoreCategoryRepository ve EfCoreProducRepository ler içinde kullanabiliriz. Her seferinde tekrarlı kodları yazmak yerine işlemleri bu sınıftan çekebiliriz. Kodlar şu şekilde olacaktır.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreProductRepository :
        EfCoreGenericRepository<Product, TSContext>, IProductRepository
    {
        public List<Product> GetPopularProducts()
        {
            using (var context =new TSContext())
            {
                return context.Products.ToList();
            }
        }
    }
}
```

Depency Injection (Data katmanında olan sınıfların Web projesinde (WebUI) kullanımı)

Data projesi içerisinde Veritabanı bağlantıları için gerekli yapıları oluşturduk. Bu yapıyı WebUI (sitemizin arayüzü olan proje) projesi içerisinde kullanalım. Bunun için öncelikle WebUI projesi içerisinde Data projesini tanıtalım. Daha öncede benzer işlemi Data projesi içerisinde WebUI projesini tanıtmıştık. Benzer şekilde burda tam tersini yapalım.



Proje başlığına sağ tuşa tıklayıp "Edit project file" işaretliğinde ilgili satırların proje dosyasına eklendiğini göreceğiz.

TS.WebUI.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Reference Include="TS.Data">
      <HintPath>..\TS.Data\bin\Debug\netcoreapp3.1\TS.Data.dll</HintPath>
    </Reference>
  </ItemGroup>

</Project>
```

Şimdi Web projesi içerisinde HomeController a gelelim. Daha önceden ilk derslerde biz basit olarak veritabanı ve ondan bilgileri getirmek için kullandığımız ProductRepository isminde sınıfı kullanmıştık. Artık bunu kullanmayacağız. Bilgilerimiz Data Projesi içerisindeki Abstract klasörü altındaki IproductRepository sınıfını kullanacağız. Hatırlayacağımız gibi bu sınıfın dolu versiyonuda EfCoreProductRepository den geliyordu böylece HomeController dosyamıza bilgiler bu uzun yolu takip ederek gelecek.

HomeController içerisine IProductRepository aşağıdaki satırlar ile injection yapalım (içerisine ekleyelim).

```
private IProductRepository _productRepository;
```

Daha sonra bir constructor satırları oluşturalım. Burada productRepository dışarıdan gönderilen nesnedir. Bunun tipi IProductRepository tipinde olacaktır.

```
public HomeController(IProductRepository productRepository)
{
    this._productRepository = productRepository;
}
```

IProductRepository çağrıldığı zaman biz bunun dolu versiyonunu çağırmamız gerekmektedir. Bu işlemi ise şöyle yapacağız. Bunun için WebUI projesi içerisindeki **Startup.cs** içindeki **ConfigureServices** metodu içerisine aşağıdaki gibi bir satır ekleyeceğiz. Burada bu proje içinde EfCoreProductRepository kullanıyoruz. İleride başka bir Repository kullanmak istersek örneğin AdoProductRepository, MySqlProductRepository gibi bu sefer onu buraya ekleyeceğiz. Böylece esnek bir katmanlı yapı kullanmış oluyoruz. Burada yaptığımız bir Interface çağırmış oluyoruz (burada IProductRepository bir interfacedir). Bunu çağırdığımızda içi dolu bir nesne gelmiş olacak (EfCoreProductRepository'de bu olmuş oluyor).

```
using TS.Data.Abstract;
using TS.Data.Concrete.EfCore;

public void ConfigureServices(IServiceCollection services)
{
       services.AddScoped<IProductRepository, EfCoreProductRepository>();

       services.AddControllersWithViews();
}
```

Şimdi veritabanı yapısını oluşturalım ve bilgi ekleme işlemleri gerçekleştirelim. Öncelikle veritabanı yapısı Entity projesi içerisindeki yapıyı kullanacağından orada birkaç düzeltme yapalım.

TS.Entity>Product.cs sınıfı içindeki aşağıdaki satırı çoğul yapılım. Tablolara arasında çoka-çok ilişkisi kuracağız.

```
public List<ProductCategory> ProductCategories { get; set; }
```

TS.Entity>Category.cs dosyası içindeki satırıda çoğul yapalım.

```
public List<ProductCategory> ProductCategories { get; set; }
```

Bu iki tablonun (Product ve Category) junction tablosunu da (ProductCategory.cs) oluşturmuştuk (sınıf kullanarak). Bu tabloda birincil anahtarları işaretlemiştik ama bunları tanıtmamıştık.

```
public class ProductCategory
 {
     public int CategoryId { get; set; }
     public Category Category { get; set; }
     public int ProductId { get; set; }
     public Product Product { get; set; }
 }
```

Buradaki birincil anahtarları Fluentapi ile bildiriyor olmamız gerekiyor. Bu işlem için Data projesi içerisindeki TSContext.cs dosyasını açalım. İçerisine aşağıdaki kodları ekleyelim. Bu metodda yapılan işlem, ProductCategory entitisine konumlanıp bu tabloya ait key bilgileri HasKey metodu ile verilmektedir. Bu tabloda key bilgileri birden fazladır (CategoryId ve ProductId).

TSContext.cs

```
using Microsoft.EntityFrameworkCore;
```

```
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class TSContext : DbContext
    {
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=TSDb");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<ProductCategory>().HasKey(c => new {c.CategoryId, c.ProductId });
        }

    }
}
```

Migration (Veritabanını sınıf yapılarına göre oluşturan class yapıları) Oluşturma

Bu işlemlerden sonra bir Migration (Oluşturulan sınıf yapılarını Veritabanına aktarma) sınıflarını oluşturalım. Bunun için komut satırından Data klasörü içine konumlanalım. Dotnet ef aracımız yüklü değilse önce onu yüklemeliyiz.

Dotnet.ef Aracını Yükleme

Bu aracın komutlarını Komut İstemi ekranından Veritabanını WebUI içinde oluşturmak için çalıştıracağız. Önce bu aracı yüklemeliyiz. Bunun için aşağıdaki şekilde Komut ekranına yükleme komutunu yazalım

(dotnet tool install –global dotnet-ef).

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet tool install --global dotnet-ef
Şu komutu kullanarak aracı çağırabilirsiniz: dotnet-ef
'dotnet-ef' aracı (sürüm '5.0.10') başarıyla yüklendi.

C:\Users\pc1\Desktop\TS.com\TS\TS.Data>
```

Yüklendiğinden emin olmak için (dotnet ef) komutunu çalıştıralım. Bir at görüntüsü ve devamında bazı bilgiler gelecektir.

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef
                  _/\__
              ---==/    \\
        ___  ___   |.    \|\
       | __|| __|  |  )   \\\
       | _| | _|   \_/ |  //|\\
       |___||_|       /   \\\/\\
```

Şimdi migrationımızı Data projesi içine konumlanmışken oluşturalım. Bu komutun son kısmında startup projemizin WebUI olduğunu söylüyoruz. Bir sonraki adımda Sqlite veritabanı webUI içinde oluşacak. (Burada “InitialCreate” adı aslında dosyaya verilen isimdir.

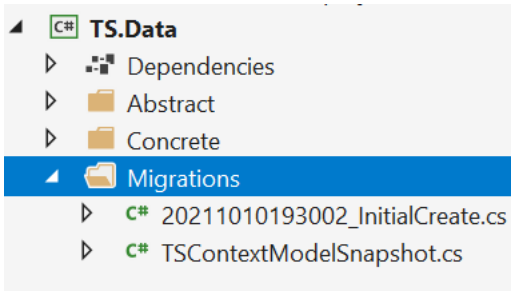
dotnet ef migrations add InitialCreate –-startup-project ../TS.WebUI

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef migrations add InitialCreate --startup-project ../TS.webUI
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Migrationımız Data projemiz içinde oluştu. (Not: migrationlarımızı kaldırdıysak -direk VS den sildiysek- sonradan bir daha oluşturmak için şu komutu kullanabiliriz

dotnet ef migrations add InitialCreate

Çünkü InitialCreate de daha önce başlangıç projesini bildirdiğimiz için belki hata verebilir. Bu şekilde bir durumla karşılaşıldı.



Oluşturduğumuz Migrations ları kaldırmak istiyorsak

dotnet ef migrations remove

komutunu TS.Data konumunda kullanabiliriz.

Migrations ları (dotnet ef database update) komutu ile çalıştırdığımızda WebUI projesi içerisinde database yapısı oluşacaktır (ama bu uygulama yanlış oldu. Bir sonraki açıklayamaya bakın).

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update
Build started...
Build succeeded.
Applying migration '20211010193002_InitialCreate'.
Done.
```

Bu oluşturduğumuz veritabanını aşağıda anlatıldığı şekilde açtığımızda içerisinde tabloları göremeyiz. Bu nedenle bu komutu çalıştırırken Startup projeyi de belirterek kullanmalıyız. Tüm solution içinde tekbir context (veritabanına bağlantı kuran dosyalar) varsa burası doğru olur fakat başka context lerde varsa o zaman hangi context kullanacağımızı da belirtmeliyiz. Yani sona örnek olarak –context TSContext gibi bir ifade eklenmelidir.

dotnet ef database update --startup-project ../TS.WebUI

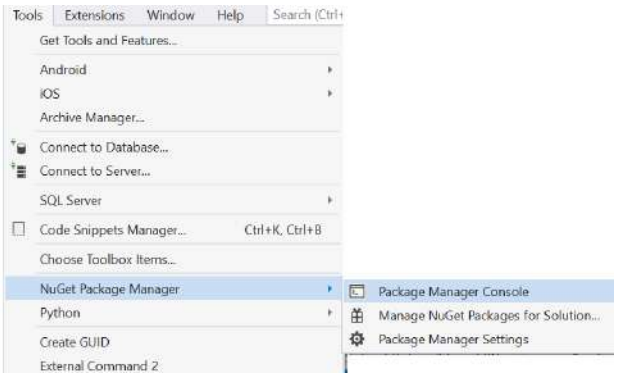
```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.WebUI
Build started...
Build succeeded.
Applying migration '20211008135252_TSMigration'.
Done.
```

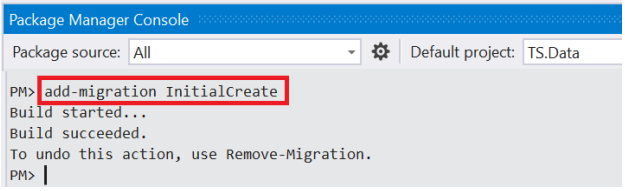
Şimdi veritabanı editörünü kurup, oluşturulan veritabanı içerisine bakalım.

Not: Data projesinin içinde Migrationlar oluşturulduktan sonra projenin Build edilmesi gerekiyor. Yoksa Veritabanı oluşsa bile içerisinde tablolar gözükmüyor.

Not: Migration oluştururken CMD satırından dotnet komutlarını kullandığımız gibi aynı zamanda VS içindeyken NuGet aracı içindeki Packace Manager Console aracını da kullanabiliriz. İlgili satıra "add-migration verilecek_isim"

şeklinde komutu kullanabiliriz. Migration kaldırmak için ise "remove-migration" komutunu kullanabiliriz. Örnek yazımı aşağıda.

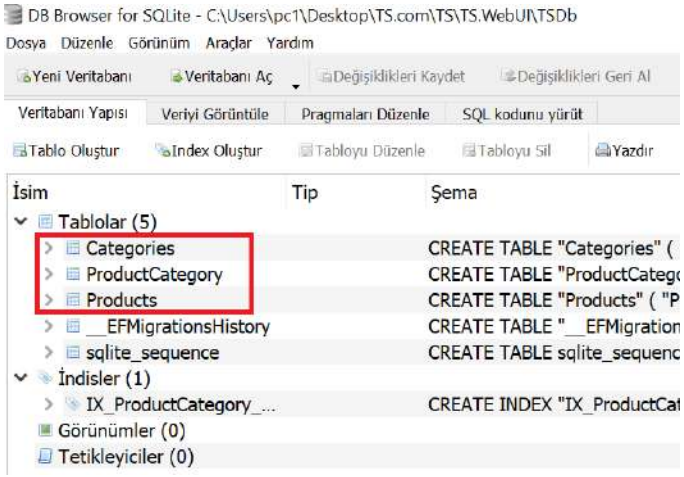




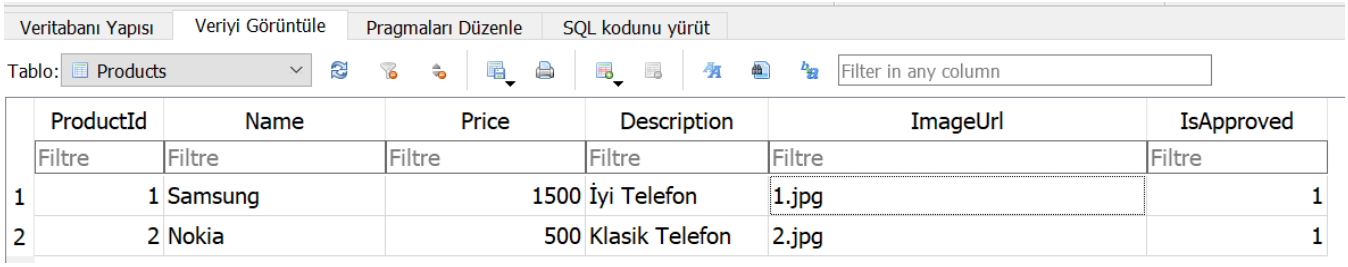
Veritabanını Editörünü Kurma

Sqlite Veritabanımız WebUI projesi içerisinde oluşturulmuştu. Bu veritabanı kullanmak için Sqlite ın internetten editorü indirip veritabanını bunda açalım. Bunun için internetten DB Browse SQLite kuruldu.

Veritabanımızı editörde açtığımızda içerisinde tablolarımızın oluştuğunu görüyoruz.



Product tablomuza iki tane bilgiyi elle ekledik.



	ProductId	Name	Price	Description	ImageUrl	IsApproved
	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre
1	1	Samsung	1500	İyi Telefon	1.jpg	1
2	2	Nokia	500	Klasik Telefon	2.jpg	1

Artık veritabanındaki bu bilgileri EfCore yapısını kullanarak sayfalarımızda görüntüleyebiliriz. Fakat bunun için önce eski oluşturduğumuz Model ve Repository yapılarını kaldırıp EfCore yapılarının referanslarını eklemeliyiz.

Data ve WebUI Projeleri içerisine gerekli Paketlerin Yüklenmesi

Data ve WebUI içinde aşağıdaki paketlerin yüklü olması gerekir. Eğer bu paketler yüklü değilse önce bunları yükleyelim. Yukarıda Migration komutları çalışmayacak olursa bu paketlerin eksik olmasından da olabilir. Bu paketleri yükledikten sonra denemeliyiz. Yukarıdaki komutlar çalışırken denemelerde bu paketler yüklüydü. Anlatımı sonradan verilmiştir.

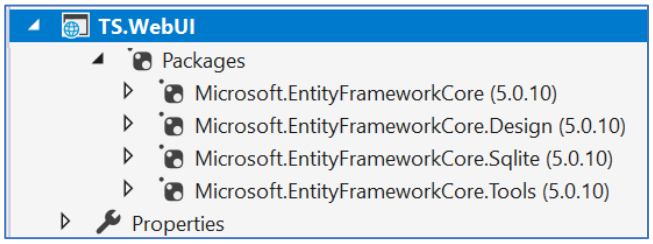
install-package Microsoft.EntityFrameworkCore

install-package Microsoft.EntityFrameworkCore.Design

install-package Microsoft.EntityFrameworkCore.Sqlite

install-package Microsoft.EntityFrameworkCore.Tools

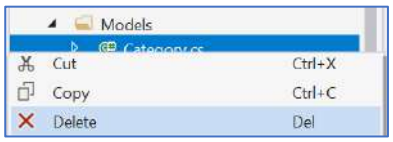
Böylece WebUI projemiz içerisinde 4 tane paketimiz olmuş oldu.



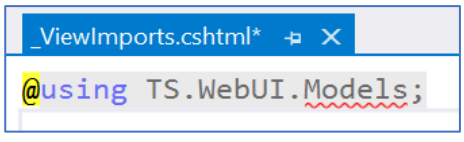
WebUI Projesi içerisindeki Eski Model ve Repository yapılarının Kaldırılması, EfCore Yapılarının Kullanıma açılması

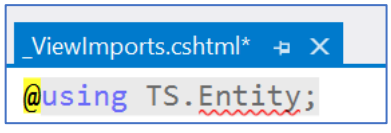
Artıp WebUI projemizin içinde daha önceden oluşturduğumuz bütün Model ve Repository yapılarını kaldırabiliriz. Çünkü artık bunların yerine TS.Entity projesini kullanacağız.

- WebUI.Model klasörü içinde oluşturduğumuz Product ve Category modellerini silelim.

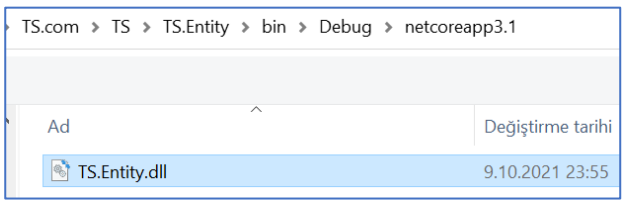


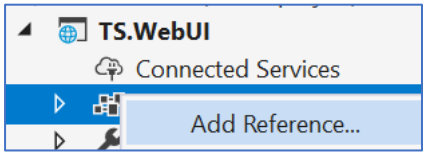
- Views klasörü içindeki _ViewImports.cshtml dosyası içinden artık model yapımızı kaldırıp onun yerine Entity projesini koyabiliriz. Fakat bunu eklediğimizde projesini görmediğini görüyoruz. Bunu tanıtalım.



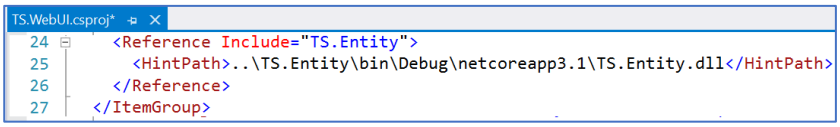


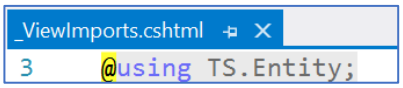
WebUI içerisine Entity projesini tanıtmak için önce Entity projesinde iken F5 basıp içerisindeki bin dosyası içinde dll yapısının oluştuğundan emin olalım. Ardından WebUI projesindeyken Depences üzerine gelip sağ tuşa tıklayıp Add Reference seçip çıkan ekrandan Browse deyip TS.Entity.dll dosyasının oluştuğu yerden seçelim.



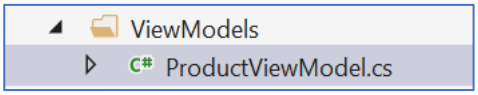


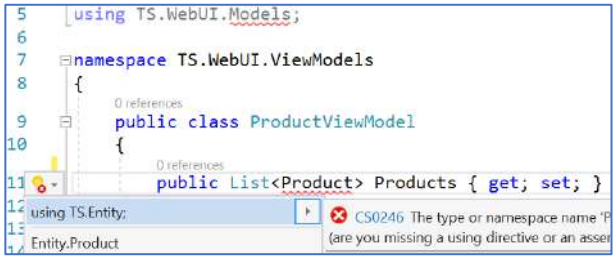
Ardından proje başlığı üzerine sağ tuşa tıklayıp "Edit Project File" linkine tıklayıp proje dosyasını açalım. Bu dosya içerisinde Entity projesinin ilgili satırlarının oluştuğunu görürüz. Artık Views içindeki _ViewImports.cshtml dosyası içinde TS.Entity projesini tanıdığını göreceğiz.



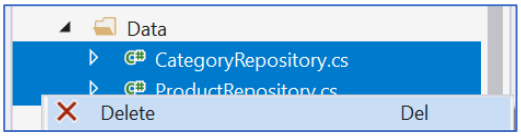


ViewModels içindeki ProductViewModel in namespace ni Entity olarak değiştirelim. Elle yazmak yerine product nesnesinin üzerinde Ctrl + (.) ya tıklarsak üstten namespace Entity olarak değiştirecektir.





- WebUI içindeki Repositoryleride kaldıralım. Çünkü bunlar bizim Data projemizin içinde vardır. Oradan gelecek.



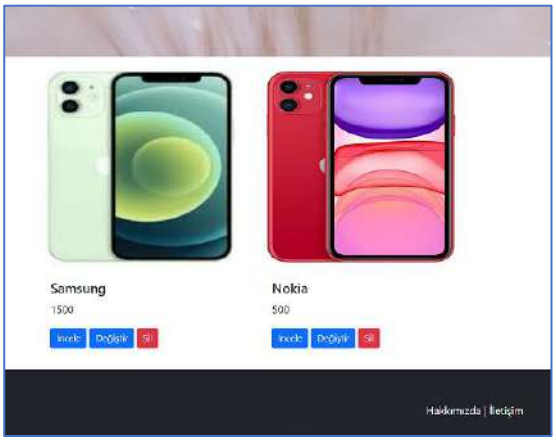
Benzer şekilde sitede hata veren tüm yerleri düzeltelim. Controllerlar içindeki metodları silmek yerine geçici olarak yorum satırı yapalım. Bunları daha sonra ayağa kaldıracağız. HomeController içinde bir çok satırı değiştirdik. ProductController içindeki metodlarda hata veren yerleri gizledik. HomeController aşağıdaki şekilde oldu. Buna göre programımız çalıştırdık.

HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TS.Data.Abstract;
using TS.Entity;
using TS.WebUI.ViewModels;

namespace TS.WebUI.Controllers
{
    public class HomeController : Controller
    {
        private IProductRepository _productRepository;

        public HomeController(IProductRepository productRepository)
        {
            this._productRepository = productRepository;
        }

        public IActionResult Index()
        {
            var Products = new List<Product>();

            //Products = ProductRepository.Products;
            Products = _productRepository.GetAll();

            return View(Products);
            //var productViewModel = new ProductViewModel()
            //{
            //    Products = _productRepository.GetAll()
            //};
            //return View(productViewModel);
        }

        public IActionResult About()
        {
```

```
            return View();
        }
    }
}
```

Projemizde gözüken iki tane ürün artık yeni oluşturduğumuz VT den gelmektedir.

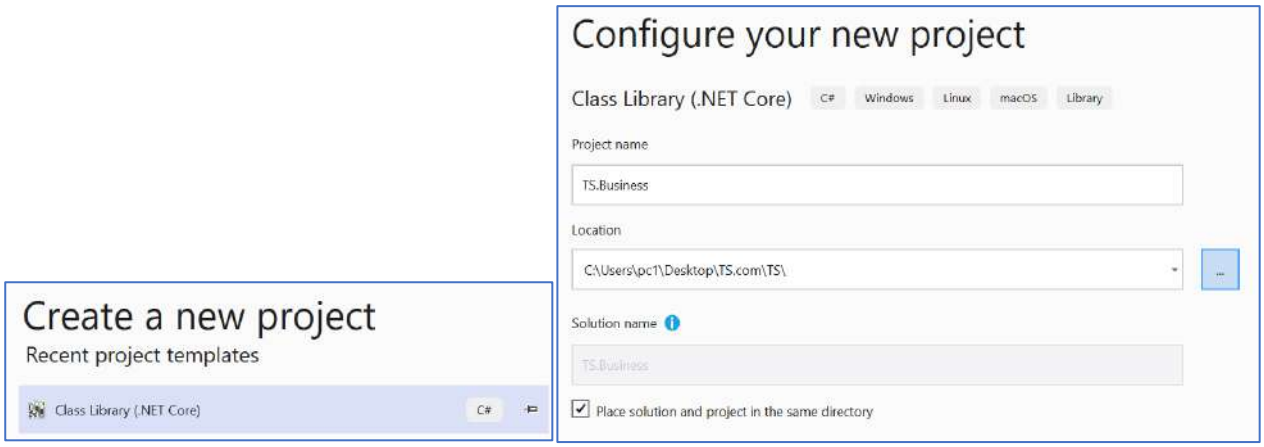


Business (İş) Katmanı

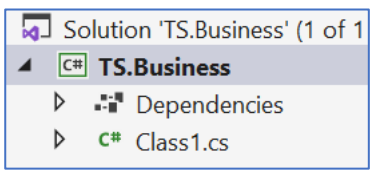
Projemiz içerisinde verileri çağırırken bir çok bilginin yapılacak işin kurallarına uygun olup olmadığını baştan kontrol edersek böylece her kullanıldığı yerde kontrol etmemize gerek kalmaz. Örneğin TCKimlik numarası 11 heneli mi? gibi. Yada bir ürünü satışa çıkarmadan önce Fiyat bilgisinin ve stokda olmasını istiyoruz. Bu gibi kontrolleri için bir iş katmanı eklemek kontrolü kolaylaştıracaktır. Böylece heryerde kontrol etmek yerine bilgiyi aldığımız esnada kontrol etmek fayda sağlayacaktır.

Bu işlem için yapacağımız uygulamalar Tıpkı WebUI, Entity ve Data gibi ayrı bir proje olacaktır. Bütün bu projeler daha üst seviye olan Solution içinde bulunacaktır.

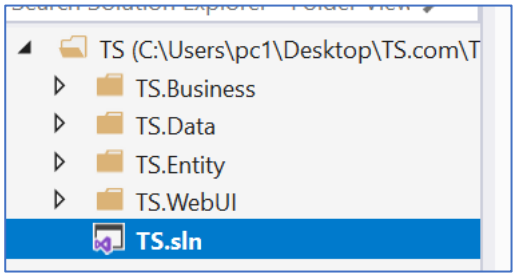
VS yu sıfırlayıp tüm projeleri kapattıktan sonra Yeni bir proje ekleme ekranına geliyoruz. Yeni projemizde tıpkı Entiyt ve Data projelerinde oluğudu gibi Class Library tipinde olacaktır.



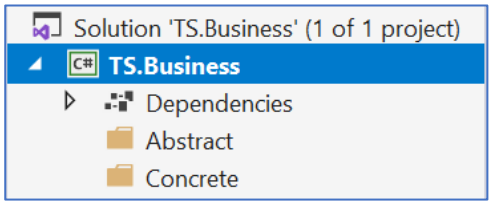
Proje gezginimiz aşağıdaki şekilde gözükecektir.



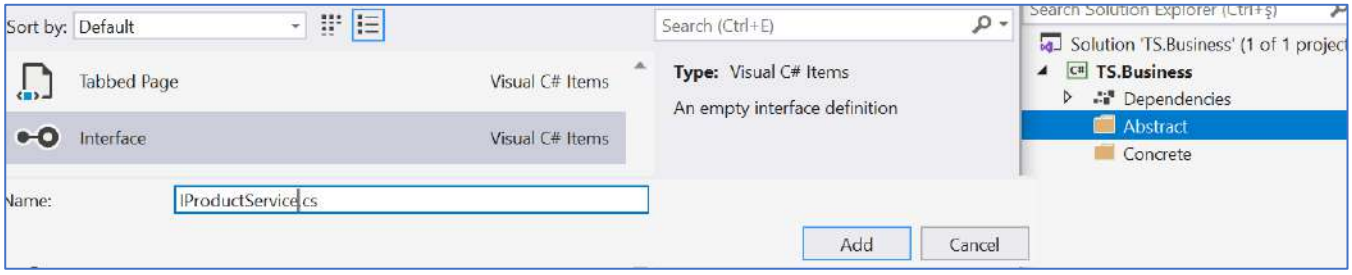
Projeyi kapatıp ana projemiz olan TS yi açtığımızda ve Klasör görünümüne geçtiğimizde 4 tane projemiz aşağıdaki şekilde gözükecektir.



Business projemizin .sln dosyasını çalıştıralım ve içerisindeki varsayılan olarak gelen Class1.cs yi kaldıralım ve tıpkı Data projesinde olduğu gibi iki tane Abstract ve Concrete adında klasörümüzü oluşturalım.

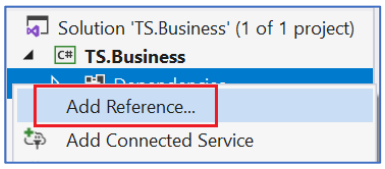


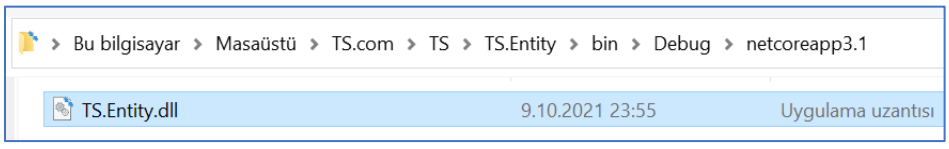
Abstract içerisinde sağ tuşa tıklayıp bir tane Interface class ı ekleyelim. Adı IproductService.cs olsun.

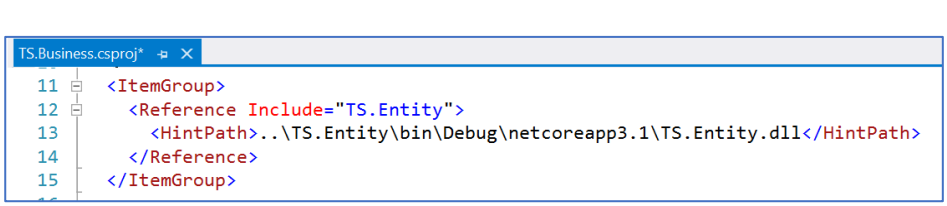


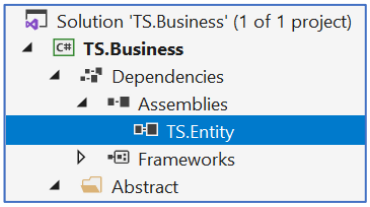
İçerisine metod tanımlamalarını Data projesi içindeki IProductRepository içinden alalım. Bu repository çoğu tanımlamasını generik (üretilebilir) olarak IRepository içinden aldığı için o sınıfın içindekileri kopyalamalıyız. Metod tanımlamalarını eklediğimizde Product nesnesini tanımadığını göreceğiz. Çünkü bulunduğumuz proje Business projesi. Oysa bu sınıf tanımlamalarımız Entity projesinden gelmektedir. O yüzden Business projesine Entity projesini tanıtmalıyız.

Bunun için Business projesi başlığına sağ tuşa tıklayıp Add Reference seçelim. Ardından çıkan ekrandan Browse butonuna tıklayıp Entity projesinin bin>Debug içindeki .dll dosyasını seçelim. Böylece Entity projesini tanıtmış olduk. Business projesinin .csproj uzantılı dosyasını açarsak içerisine Entity projesinin eklendiğini görürürüz. Bu dosyayı açmak için Business proje başlığı üzerine gelip "Edit Project File" seçebilirsiniz.



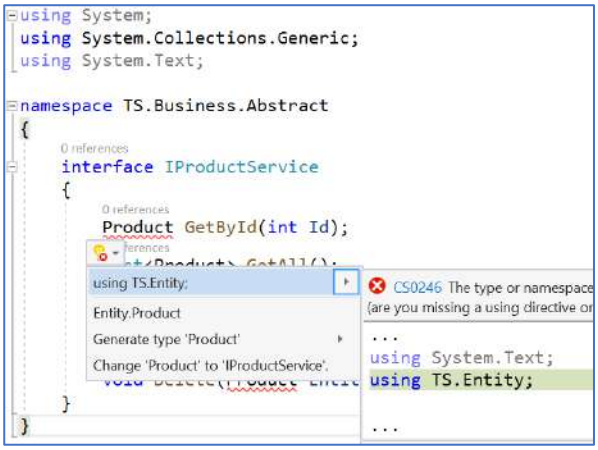


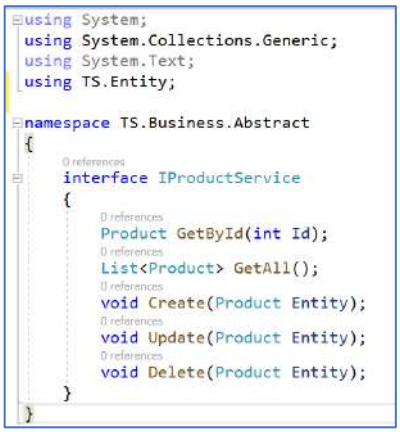




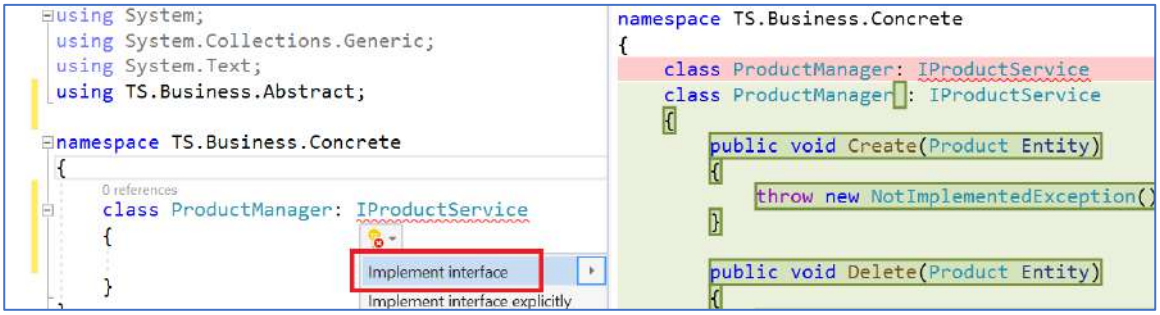
Benzer şekilde Business projesi Data projesine de ihtiyaç duyacağından onun Reference sını da ekleyelim. Çünkü Context ler Data projesi içerisinde Product ve Category sınıfları ise Entity projesi içerisindedir.

Artık IproductService dosyamızın içindeki Product nesnelerini tanıtmak için Entity namespace ni yukarı ekleyebiliriz.





Şimdi İş katmanındayken Concrete klasörünün içerisinde bir Class oluşturalım. İsmini de ProductManager.cs verelim. Bu sınıf içerisine ilgili Product service metodlarını imlement edelim. Böylece sınıf içerisine veritabanı metodları içi boş bir şekilde gelecektir.



Biz ProducManager ı niçin kullanacağız. Örneğin biz ürün eklemeden önce Burada Create metodu içerisinde bir işlem yapacağız. Gerekli şartları sağladıktan sonra da Data projesi içindeki EfCoreProductRepository kullanarak da veritabanına işlemi gerçekleştireceğiz. Yani veritabanına sorguyu göndermeden önce belli kriterleri sağlayıp sağlamadığını kontrol edeceğiz. Ön bir denetim işlemi yapmış olacağız. Buna göre Product entity si için oluşturduğumuz manager sınıfı aşağıdaki şekilde olacaktır.

Bu manager Product entity si için kullanılacak. Projemizde bir çok entitiler olabilir (Product, Category, Order gibi) Her biri için ayrı bir ProductManager yazdığımız gibi ayrı bir manager yazmamız gerekir. Ayrıca dikkat edersek burada kullandığımız EfCoreProductManager sınıfı yerine başka bir veritabanı erişim teknolojisine geçilmiş olabilir. Böyle bir durumda bütün bu Managerlar içindeki EfCoreProductManager sınıf isimlerini değiştirmemiz gerekecektir. O sebeple buradaki EfCoreProductManager sınıfının olduğu yeri daha Generic, hepsi için değiştirmeden kullanabileceğimiz tarzda yazalım.

```
using TS.Data;
using TS.Data.Concrete.EfCore;
using TS.Entity;

namespace TS.Business.Concrete
{
    0 references
    class ProductManager : IProductService
    {
        EfCoreProductRepository productRepository = new EfCoreProductRepository();

        1 reference
        public void Create(Product entity)
        {
            //İş Kuralları Buraya yazılacak
            productRepository.Create(entity);
        }

        1 reference
        public void Delete(Product entity)
        {
            //İş Kuralları Buraya yazılacak
            productRepository.Delete(entity);
        }
```

ProductManger dosyasını Generic oluşturma

Yukarıdaki gibi EfCoreProductRepository kullandığımızda ileride başka bir Veritabanı sistemine geçtiğimizde buradaki kodların hepsini değiştirmemiz gerekecektir. Biz burayı daha generic yapmak istersek yani veritabanı sistemiz değişse bile burada bir değişikliğe ihtiyaç duymaması için EfCoreProductRepository nin imlement edildiği sınıf üzerinden yazmamız gerekir. EfCoreProductRepository ise IProductRepository üzerinden imlement edildiği için o sınıfı kullanmalıyız. Böylece EfCoreProductRepository ye bağımlılık kalkmış olacaktır. Yani Interface üzerinden işlem yapmış olacağız.

Bunun için aşağıdaki kodları sınıfımızın başına ekleyelim.

```
class ProductManager : IProductService
{
    private IProductRepository _productRepository;

    public ProductManager(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
```

Bu kodlar sayesinde dışarıdan gelen productRepository nesnesi IProductRepository tipinde olmak zorundadır. Bu nesne bu sınıfın içinde kullanılacak olan _productRepository içine atılarak kullanılacaktır. Böylece bu repository sınıfının hangi veri erişim teknolojisini (örneğin EfCore gibi) kullandığı bizi ilgilendirmemiş olacak. ProductManager sınıfımız bir sanal sınıfla çalışıyor.

ProductManager.cs

```
using System.Collections.Generic;
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Business.Concrete
{
    public class ProductManager : IProductService
    {
        private IProductRepository _productRepository;
        public ProductManager(IProductRepository productRepository)
        {
            _productRepository = productRepository;
        }
        public void Create(Product entity)
        {
            // iş kuralları buraya eklenecek
            _productRepository.Create(entity);
        }

        public void Delete(Product entity)
```

```
        {
            // iş kuralları buraya eklenecek
            _productRepository.Delete(entity);
        }

        public List<Product> GetAll()
        {
            return _productRepository.GetAll();
        }

        public Product GetById(int id)
        {
            throw new System.NotImplementedException();
        }

        public void Update(Product entity)
        {
            throw new System.NotImplementedException();
        }
    }
}
```

---------- 000----------

Sanal sınıfla çalışırken IProductManager çağrıldığında ProductManager ın gönderilmesi gerekiyor. Peki bu çağırma işlemi nerede yapılacak. Tabii ki WebUI projesinde yapılacaktır. WebUI projesi içerisinde StartUp.cs ye gelelim. Bu dosya içinde daha önce eklediğimiz satırları bulalım. Daha önceden IProductRepository sanal sınıfını çağırdığımızda onun dolu versiyonu olan EfCoreProductRepository göndermesi için satırlar yazmıştık. Benzer şekilde aynı yere yeni kullanacağımız sanal sınıfın dolu versiyonunu göndermesi için de aşağıdaki kodları yazalım.

```
namespace TS.WebUI
{
    1 reference
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit https://go.microsoft.com
        0 references
        public void ConfigureServices(IServiceCollection services)
        {
            //Data projesi içerisindeki IProductRepository çağrıldığında onun dolu versiyonu olan Ef
            services.AddScoped<IProductRepository, EfCoreProductRepository>();

            services.AddScoped<IProductService, ProductManager>();

            //1.İŞLEM: Bu yazı MCV deseni kullanımı için Controller Kullanımını Aktif ediyor.
            services.AddControllersWithViews();

        }
```

Fakat kodların çalışabilmesi için TS.WebUI projesinin TS.Manager projesini tanıyor olması gerekiyor. Yani Reference nı eklemeliyiz. Daha önce yukarılarda benzer uygulamaları yaptık. O şekilde ekleyelim (Depencess üzerine gelip sağ tuşa tıklayıp Add Reference seçiyoruz ve Manager projesi içinde Bin>Debug içindeki .dll dosyasını ekliyoruz. Dosyanın üst kısmında ise aşağıdaki namespacelerinde eklenmiş olması gerekir.

```
using TS.Business.Abstract;
using TS.Business.Concrete;
```

Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using TS.Data.Abstract;
using TS.Data.Concrete.EfCore;
using TS.Business.Abstract;
using TS.Business.Concrete;
```

```
namespace TS.WebUI
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            //Data projesi içerisindeki IProductRepository çağrıldığında onun dolu versiyonu olan EfCoreProductRepository
gönderecek.
            services.AddScoped<IProductRepository, EfCoreProductRepository>();

                //IProductService çağrıldığında, ProductManager ı gönderecek.
                services.AddScoped<IProductService, ProductManager>();

            //1.İŞLEM: Bu yazı MCV deseni kullanımı için Controller Kullanımını Aktif ediyor.
            services.AddControllersWithViews();

        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseStaticFiles(); //wwwroot içindeki statik dosyaların okunabilmesi için konuldu

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {

                //2.İŞLEM: Adres Deseni için bu format eklendi.
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=home}/{action=index}/{id?}"
                    );


            });
        }
    }
}
```

Artık bizim HomeController ımız IProductRepository ile değil IProductService ile çalışacak. Araya birde iş kurallarını ekliyoruz. HomeController ımız artık tamamen iş katmanı ile çalışacak. HomeController içerisini aşağıdaki şekilde değiştirelim.

HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.WebUI.Controllers
{
    public class HomeController : Controller
    {
        private IProductService _productService;

        public HomeController(IProductService productService)
        {
            this._productService = productService;
        }

        public IActionResult Index()
        {
            var Products = new List<Product>();

            //Products = ProductRepository.Products;
           Products = _productService.GetAll();
```

```
            return View(Products);

            //var productViewModel = new ProductViewModel()
            //{
            //    Products = _productRepository.GetAll()
            //};

            //return View(productViewModel);
        }

        public IActionResult About()
        {
            return View();
        }
    }
}
```

Startup.cs içindeki aşağıdaki satırdan anladığımız, biz IProductService çağırıyoruz. Bunu çağırdığımızda oda ProductManager ı çağırmış olacak.

```
services.AddScoped<IProductService, ProductManager>();
```

ProductManager çalıştığında ise bu dosya içinde bir aşağıdaki gibi injection işlemi var. Bu işlemde ise IProductRepository kullanılıyor.

```
private IProductRepository _productRepository;
0 references
public ProductManager(IProductRepository productRepository)
{
    _productRepository = productRepository;
}
```

Startup.cs dosyasının içine baktığımızda burada IProductRepository çağırıldığında ise EfCoreProductRepository çağrılmış olmaktadır.

```
services.AddScoped<IProductRepository, EfCoreProductRepository>();
```

Yani biz burada sanalı çağırıyoruz ve onun dolu versiyonu olan Concrete versiyonu geliyor. Business katmanı ekleyerek bilgileri direk data katmanından almak yerine araya bir kontrol işlemi yapmak Business katmanından bilgileri almış oluyoruz.

Bundan sonraki bölümde Veritanımıza belli sayıda bir test verisi ekleyelim.

Dikkat: bir projedeki class ı başka bir projede kullanırken başına **public** ifadelerini eklemeyi unutmayın!

Dikkat: Bir projeyi başka bir projeye referans olarak gönderirken **Build** etmeyi unutmayın! Dll ler oluşsun.

Test Verilerinin Otomatik Eklenmesi

Bu bölümde Veritabanına bir metod üzerinden verileri nasıl ekleyeceğimizi görelim. Bu işlem için Data>Concret>EfCore içerisine yeni bir Class.cs ekleyelim. Bu sınıfın adı **SeedDatabase** isminde statik bir sınıf olsun. İçerisinde statik bir metod olsun ve geri dönüş değeri olmasın ve ismi de Seed() olsun. İçerisinde TSContex sınıfından bir nesne türetelim ve burada contex üzerinden veritabanı kontrol edelim.

Bu metod içinde bekleyen migration (veritabanına aktarılmamış yapılar) var mı kontrol edelim. Bu işlem için migrationların sayısına bakalım. Bu işlem için aşağıdaki if yapısını ekledik. Fakat bu if yapısı içindeki GetPendingMigrations() metodunun çalışabilmesi için yukarı (`using Microsoft.EntityFrameworkCore;`) eklenmesi ve .Count() metodunun çalışabilmesi içinde (`using System.Linq;`) kütüphanelerinin üste eklenmesi

gerekir. GetPendingMigrations() ifades migrationların isimlerini bir liste şeklinde getirecektir fakat biz ismiyle ilgilenmiyoruz sayısına bakıyoruz. Burada .Count() metodu bu listedeki elemanların sayısını getirir ama onun kütüphanesi ayrı bir isimdir yani Sistem altında çalışan Linq kütüphanesidir.

```
if(context.Database.GetPendingMigrations().Count()==0)
```

Şimdi bu if yapısı içerisinde eğer veritabanında Categories yoksa onları ekleyelim.

```
if(context.Categories.Count()==0)
{
        context.Categories.AddRange();
}
```

Hemen aşağısında geriye bir Category listesi gönderen statik bir metod oluşturalım. Category[] kategori üzerine tıklayalım ve Entity namespacesini (using TS.Entity;) yukarı ekleyelim. Yada burası bir metod olmak yerine bir properties şeklinde olsun (Yazım şekline dikkat edin). Bu properties içerisine birkaç kategori ekleyelim.

```
private static Category[] Categories ={
        new Category() {Name="Telefon"},
        new Category() {Name="Bilgisayar"},
        new Category() {Name="Elektronik"}
};
```

Bu kategori listesini AddRange metodu ile context içindeki kategorilere ekleyebiliriz.

```
context.Categories.AddRange(Categories);
```

Benzer şekilde bu seferde kategori yerine Producs ları soralım. Eğer eklenmemişse onları ekleyelim. Categories için yaptığımız liste ekleme işlemlerini Products içinde yapalım. Oluşturduğumuz static sınıfın son hali şu şekilde olmuş olacaktır.

SeedDatabase.cs

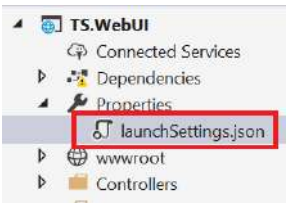
```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public static class SeedDatabase
    {
        public static void Seed()
        {
            var context = new TSContext();

            if(context.Database.GetPendingMigrations().Count()==0)
            {
                if(context.Categories.Count()==0)
                {
                    context.Categories.AddRange(Categories);
                }

                if (context.Products.Count() == 0)
                {
                    context.Products.AddRange(Products);
                }
            }
            context.SaveChanges();
        }

        private static Category[] Categories ={
            new Category() {Name="Telefon"},
            new Category() {Name="Bilgisayar"},
            new Category() {Name="Elektronik"}
        };
```

```
        private static Product[] Products ={
            new Product() {Name="Samsung S5", Price=1000, ImageUrl="1.jpg", Description="İyi Telefon",
IsApproved=true},
            new Product() {Name="Samsung S6", Price=2000, ImageUrl="2.jpg", Description="İyi Telefon",
IsApproved=false},
            new Product() {Name="Samsung S7", Price=3000, ImageUrl="3.jpg", Description="İyi Telefon",
IsApproved=true},
            new Product() {Name="Samsung S8", Price=4000, ImageUrl="4.jpg", Description="İyi Telefon",
IsApproved=false},
            new Product() {Name="Samsung S9", Price=5000, ImageUrl="5.jpg", Description="İyi Telefon",
IsApproved=true}
        };

    }
}
```

Artık elimizde veritabanına eklenmeyi bekleyen statik bir sınıfımız var. Bu sınıfı projenin herhangi bir aşamasında veritabanına eklemeliyiz. Bu ekleme içini Startup.cs içindeki IsDevelopment() metodu içeirisinde yapabiliriz. Çünkü proje daha geliştirilirken burası çalışacak anlamındadır. SeedDatabase sınıfının çalışabilmesi için üst kısımda onun namespacenin olması gerekir (`using TS.Data.Concrete.EfCore;`) Eğer yinede hata veriyorsa yukarıda EfCore içinde oluşturduğumuz SeedDatabase sınıfının derlenmiş hali oluşmamıştır. Yani projeyi Build etmemiz gerekir.

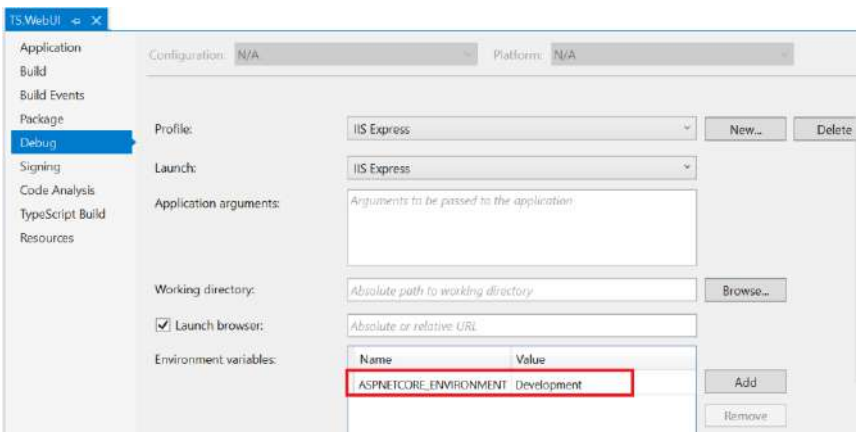
```
if (env.IsDevelopment())
{
        SeedDatabase.Seed();

        app.UseDeveloperExceptionPage();
}
```

Peki buradaki IsDevlopment() metodu bizim henüz projeyi geliştirme aşamasında olduğumuz nasıl anlıyor. WebUI projesi içerisindeki Properties klasörü içindeki LaunchSettings.json ismindeki dosyada yazılı olan Environment ifadesi içindeki “Development” ifadesinden anlıyor. Bu ifade IsDevelopment kısmından True getirecektir. Biz projemizi yayınladığımız zaman ise burayı “**Production**” na dönüştüreceğiz.

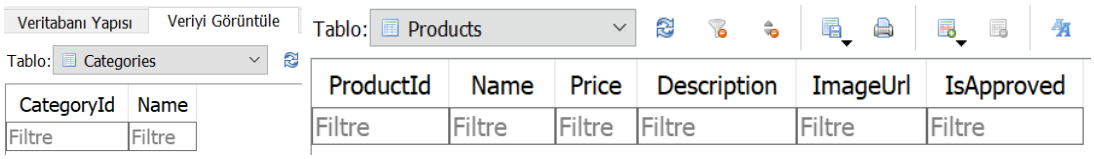



```
"TS.WebUI": {
  "commandName": "Project",
  "launchBrowser": true,
  "applicationUrl": "https://localhost:5001;http://localhost:5000",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
```

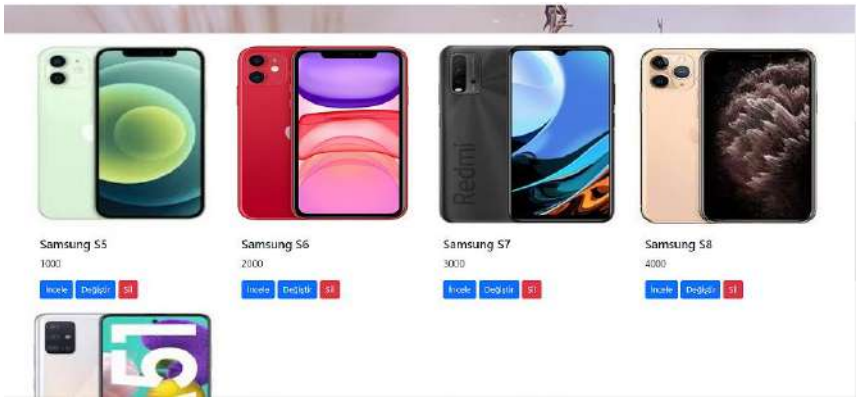
Aynı değişiklikleri Proje adında sağ tuşa tıklayıp Properties penceresini açarsak ve buradan Debug kısmından ayarlayabiliriz.



Şimdi veritabanını açalım içerisindeki verileri boşalt.



Projeyi çalıştıralım. 5 tane ürün eklemiştik. Bunlar gelecektir.



Şu ana kadar proje alt yapımız hazırdır. Bir sonraki bölümde ise kullanıcı ve yönetici sayfalarını oluşturmaya başlayalım.

Kurs Kaynağı: Udemy-Komple Uygulamalı Web Geliştirme Kursu-Sadık Turan