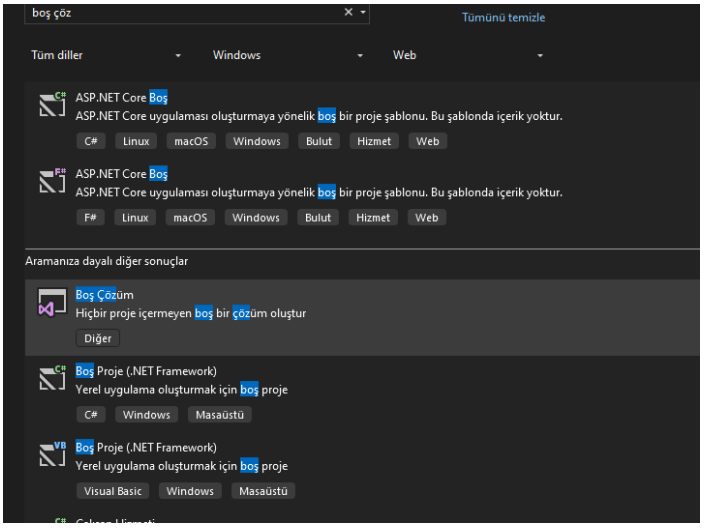
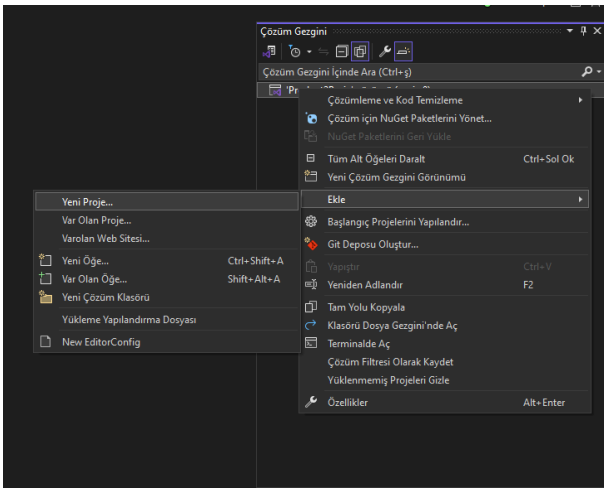
MVC 5 PROJE GELİŞTİRME

1.ADIM: Projemizi oluşturuyoruz.

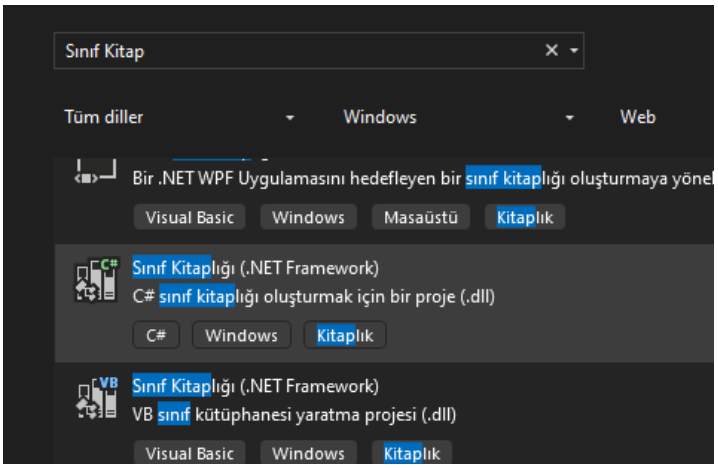
- boş çözüm (solition blank) açılıyor.



- Boş solition üzerine sağ tıklayıp katmanları ekliyoruz

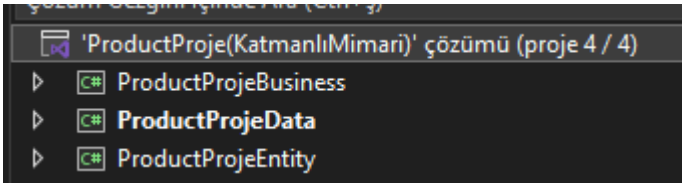


- Oluşturacağımız katmanlar:
    - DATA : Veri tabanı ile bağlantımızı sağlıyoruz.
    - Entity: Nesnelerimi(sınıflarımızı) oluşturuyoruz.(Product,Category..)
    - Business: Asıl işlemlerin gerçekleştiği katman
    - UI : Kullanıcı ile iletişime geçtiğimiz arayüzlerin oluştuğu katman

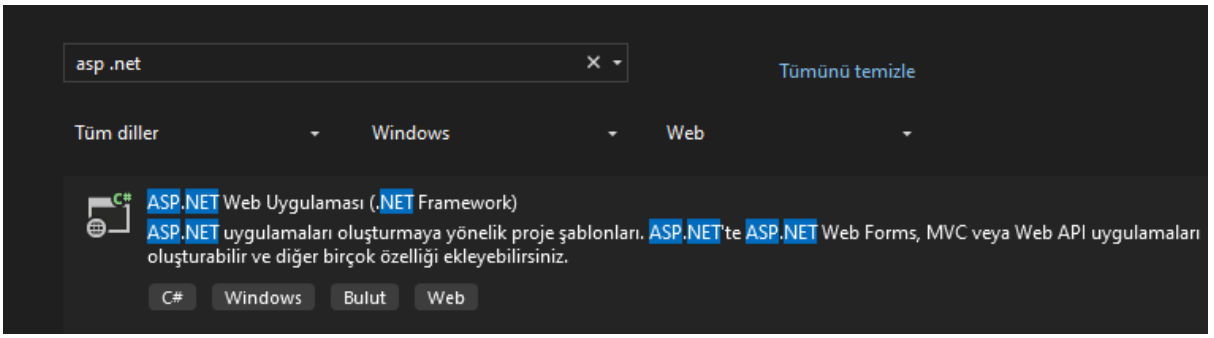


- “Class Library” oluşturuyoruz (.NET FRAMEWORK) şeklinde

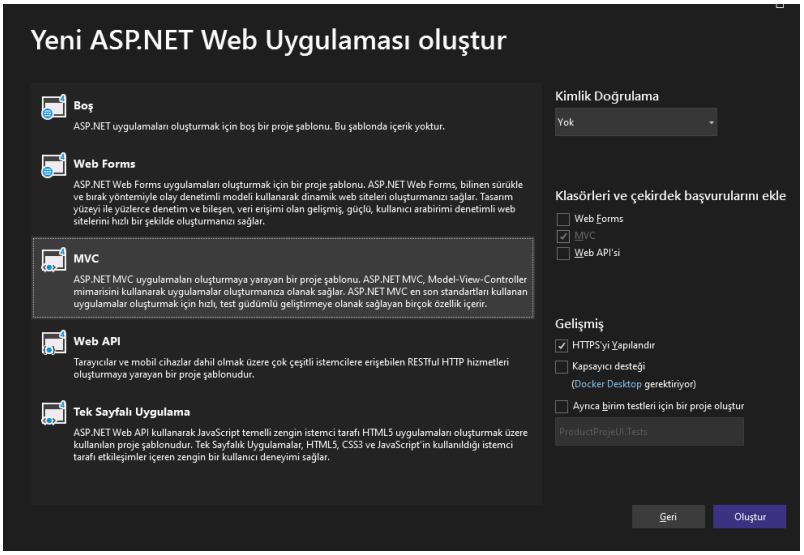
3 katmanımızı ekliyoruz.



- UI KATMANIMIZI mvc olarak ekliyoruz

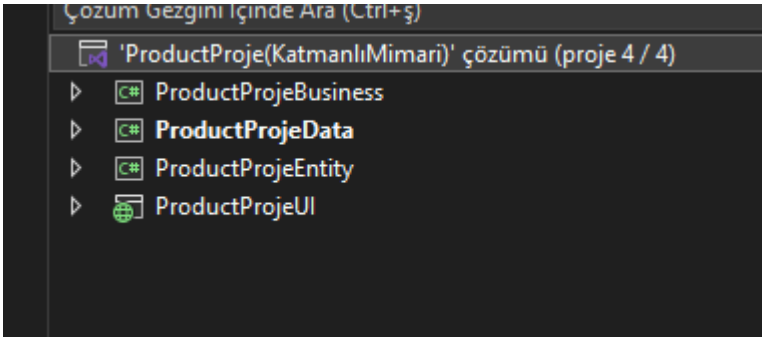


eklerken:



MVC Seçiyoruz çünkü elimizde hazır template ler gelsin istiyoruz.

Böylece tüm katmanlarımız oluşmuş oldu.



Burada Business,Data, Entity katmanındaki hazır gelen class larımızı siliyoruz.

2 .ADIMI ENTİTY KATMANI

entity klasörü oluşturup içine Product.cs,Category.cs,Order.cs oluşturuyoruz.

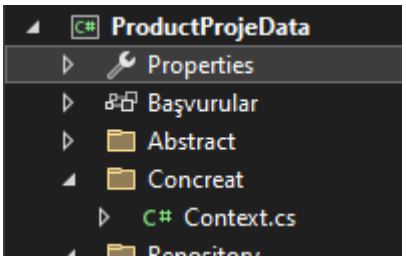
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProductProjeEntity.Entity
{
    2 başvuru
    public class Category
    {
        0 başvuru
        public int CategoryID { get; set; }
        0 başvuru
        public string CategoryName { get; set; }
        0 başvuru
        public string Description { get; set; }
        0 başvuru
        public ICollection<Product> Products { get; set; }// Category ile Product sınıfını bağladık
    }
}
```

```
{
    3 başvuru
    public class Product
    {
        0 başvuru
        public int ProductID { get; set; }
        0 başvuru
        public string ProductName { get; set; }
        0 başvuru
        public int Stock { get; set; }
        0 başvuru
        public decimal Price { get; set; }
        0 başvuru
        public int CategoryID { get; set; }
        0 başvuru
        public int OrderId { get; set; }
        0 başvuru
        public virtual Order Order { get; set; } // her ürünün bir siparişi olabilir
        0 başvuru
        public virtual Category Category { get; set; } // her ürünün bir kategorisi olabilir

    }
```

```
{
    2 başvuru
    public class Order
    {
        0 başvuru
        public int OrderID { get; set; }
        0 başvuru
        public int ProductID { get; set; }
        0 başvuru
        public ICollection<Product> Products { get; set; }
        // Bir siparişin (Order), birden fazla ürünü (Product) olabilir.


    }
```

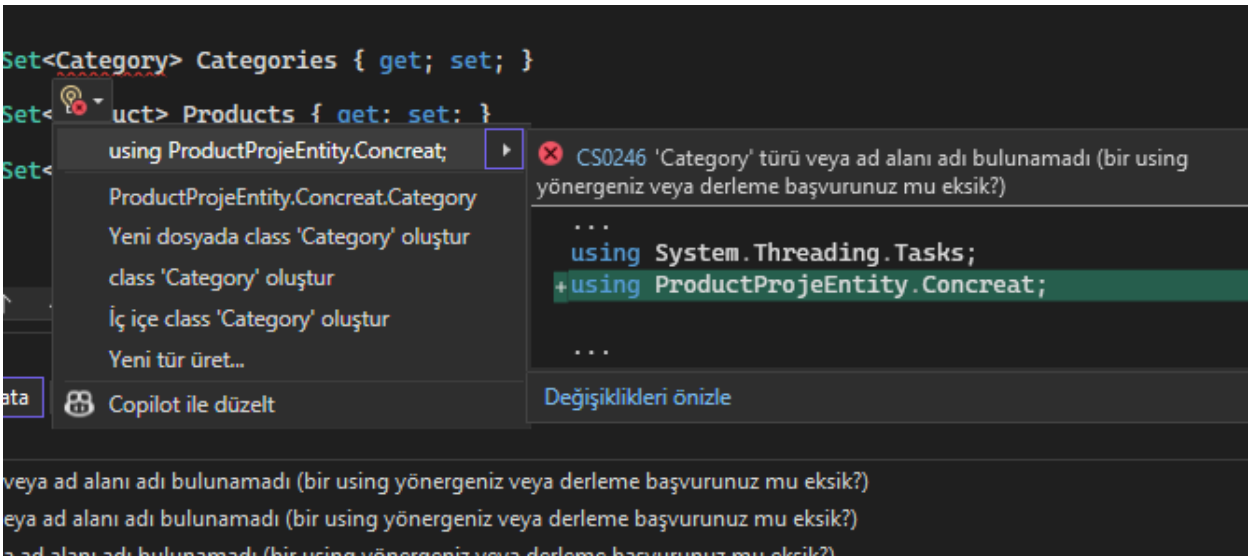
3 .ADIM DATA KATMANI

Context.cs oluşturacağız bunun için concreat diye klasör açıyoruz ve içine Contex.cs oluşturuyoruz.

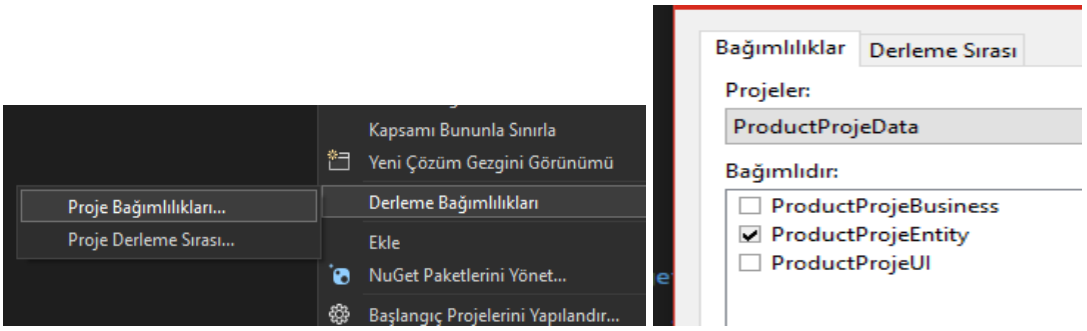



```
public class Context:DbContext
{
    0 başvuru
    public DbSet<Category> Categories { get; set; }
    0 başvuru
    public DbSet<Product> Products { get; set; }
    0 başvuru
    public DbSet<Order> Orders { get; set; }

}
```

Context:DbContext eklediğimizde yukarıya otomatik using şeklinde kütüphane eklenir. eğer eklenmezse entity framework kütüphanesini eklemeniz gerekir projenize.
Entityden referans göstermen lazım referans gösterdiğimiz zaman
using ProductProjeEntity.Concreat; eklenir.



- Referans işi ile uğraşmamak için ProductProjeData yasağ tıklayıp ReferanceManager(Derleme Bağımlılıkları) diyerek ProductProjeData katmanının nerden referans alacağını seçebiliyoruz.

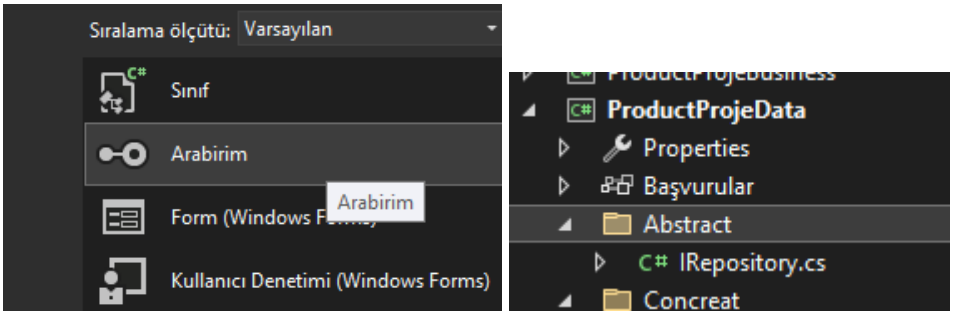


4.ADIM DATA KATMANINDA INTERFACELER

Prensiplere göre şimdi projenin içerisinde soyut yapıları tutacak interface yapılara ihtiyacımız var. (Bunlar için Abstract klasörü oluşturuyorum)

3 Nesne için (Product,Category,Order) ortak operasyonlar olacak(Örneğin Kaydetme, Silme, Güncelleme)
yani bir kez oluşturup 3 nesne de de bu işlemleri kullanabileceğim.

ProductProjeData dan Abstract klasörü oluşturduktan sonra new item → Interface(Arabirim) oluşturuyoruz



IRepository.cs adında bir Arabirim oluşturuyoruz. Ayrı ayrı oluşturmamak için bunu 3 nesne de kullanabileceğiz.

NOT: **somut** yapıları (product gibi) Concreat klasöründe, **soyut** yapıları ise (Delete) Abstract klasöründe tutuyoruz.

```
namespace ProductProjeData.Abstract
{
    //Veritabanı işlemleri için ortak bir yapı oluşturuyoruz.
    // bir Generic (Genel) Repository Arayüzü tanımlar.
    1 başvuru
    public interface IRepository<T>   //T yerine Product Category Order kullanılabilir.
    {
        1 başvuru
        List<T> List(); //Veritabanındaki tüm kayıtları getirir.
        1 başvuru
        T Get(Expression<Func<T, bool>> filter); //Belirtilen bir şarta göre tek bir kayıt döndürür.
        1 başvuru
        void Insert(T entity);
        1 başvuru
        void Update(T entity);
        1 başvuru
        void Delete(T entity);
        1 başvuru
        List<T> List(Expression<Func<T, bool>> filter);//Belirli bir şarta göre birden fazla kayıt döndürür.
    }
}
```

Interface yapılarında gövde, erişim belirleyicisi yer almaz. Bu yüzden işlemin nasıl yapılacağını söylemeyiz sadece işlemi tanımlıyoruz. (Insert, Update..)

- Bu yukarıdaki operasyonların nasıl çalışacağını bildirmemiz gerekiyor. Bunun için Dataya Repository adında klasör oluşturuyoruz.
  GenericRepository.cs adında bir class ekliyoruz.

```
namespace ProductProjeData.Repository
{
    1 başvuru
    public class GenericRepository<T> : IRepository<T> where T : class

    {
        Context db = new Context();
        DbSet<T> _object; // _object veri tabanı tablosunu temsil ediyor


        0 başvuru
        public GenericRepository()//ctor tab tab yaparak oluşturuyoruz
        {
            _object = db.Set<T>(); //db den gelen değeri object e aktarmak için kullanıyoruz
        }


        1 başvuru
        public void Delete(T entity)
        {
      //Entity Framework kullanarak bir nesneyi (entity) veritabanından silmek için yazılmıştır
      var sil = db.Entry(entity); // entity nesnesinin veritb.daki durumunu takip için bir EntityEntry nesnesi döndürür.
        // Yani veritabanındaki hangi durumda olduğunu(Added, Modified, Deleted, Unchanged) anlamamızı sağlar.
        sil.State = EntityState.Deleted;// Bu satır, entity nesnesinin silinmesi gerektiğini belirtiyor.
        db.SaveChanges();
        }

        1 başvuru
        public T Get(Expression<Func<T, bool>> filter)
        {
            return _object.SingleOrDefault(filter);
        }
```

```
        1 başvuru
        public void Insert(T entity)
        {
          var ekle = db.Entry(entity);
            ekle.State = EntityState.Added;
            db.SaveChanges();
        }

        1 başvuru
        public List<T> List()
        {
            return _object.ToList();
        }

        1 başvuru
        public List<T> List(Expression<Func<T, bool>> filter)
        {
            return _object.Where(filter).ToList();
        }

        1 başvuru
        public void Update(T entity)
        {
          var guncelle = db.Entry(entity);
            guncelle.State = EntityState.Modified;
            db.SaveChanges();
        }
    }
}
```

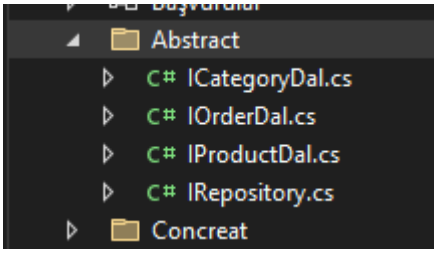
Burada her nesne için ayrı ayrı sınıf yazmak yerine veritabanı işlemlerini tek bir yerde tutuyorum.

Kısaca ;
IRepository de ortak operasyonlar oluşturdum nesne alacağımı bildirdim.
GenericRepositoryden bu operasyonların nasıl işleneceğini bildirdim.
IRepositoryde T ile sınıfın hangisi olduğunu bildiriyoruz.

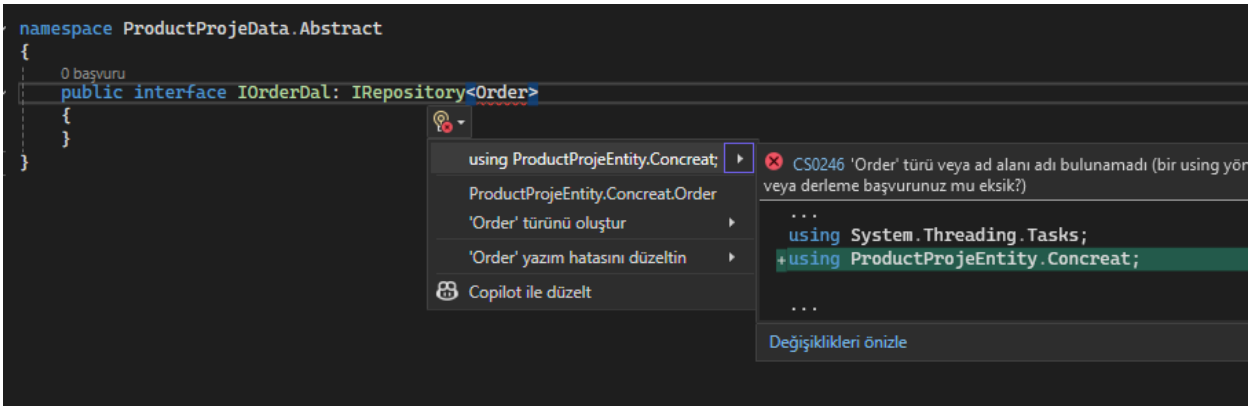
> `GenericRepository<T>` sınıfının `IRepository<T>` arayüzünden miras alması, **IRepository'de tanımlanan metodların GenericRepository'de uygulanması (implement edilmesi) gerektiği anlamına gelir**.

5. ADIM

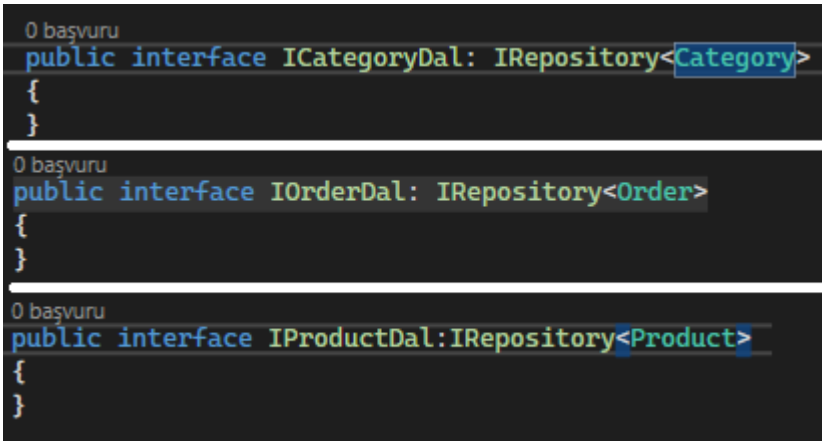
Abstract klasörünün içinde 3 tane Interface ekliyoruz her bir sınıf için



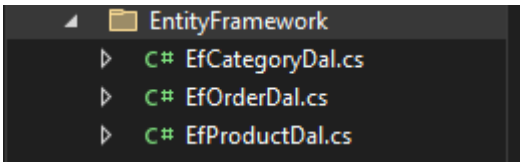
Eklerken referans aldığımız yeri using olarak da eklemeyi unutmuyoruz örneğin:



her birini IRepository e ile bağlıyoruz.



- Daha sonra EntityFramework klasörü açıp Datanın içinde 3 nesneme sınıf açıyorum

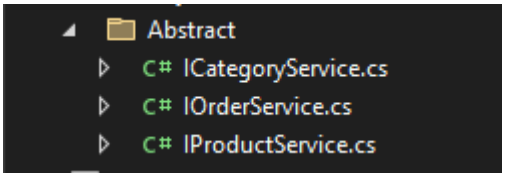


```
0 başvuru
public class EfProductDal:GenericRepository<Product>, IProductDal
{
}

0 başvuru
public class EfOrderDal:GenericRepository<Order>, IOrderDal
{
}

0 başvuru
public class EfCategoryDal:GenericRepository<Category>, ICategoryDal
{
}
```

İçlerini de bu şekilde sınıflarını ve geldikleri GenericRepository belirtiyoruz.

6. ADIM BUSINESS KATMANI 👍

Abstract klasörü oluşturup ICategoryService, IProductService, IOrderService ekliyoruz interface olarak.



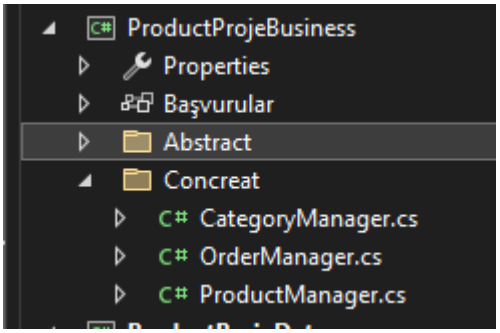

```
public interface ICategoryService
{
    1 başvuru
    List<Category> List();
    0 başvuru
    void CategoryInsert(Category entity);
    0 başvuru
    void CategoryUpdate(Category entity);
    0 başvuru
    void CategoryDelete(Category entity);

}
```

```
0 başvuru
internal interface IOrderService
{
    0 başvuru
    List<Order> List();
    0 başvuru
    void OrderInsert(Order entity);
    0 başvuru
    void OrderUpdate(Order entity);
    0 başvuru
    void OrderDelete(Order entity);
}
```

```
0 başvuru
internal interface IProductService
{
    0 başvuru
    List<Product> List();
    0 başvuru
    void ProductInsert(Product entity);
    0 başvuru
    void ProductUpdate(Product entity);
    0 başvuru
    void ProductDelete(Product entity);
}
```

IRepository operasyonlarını buraya ekliyorum, fazlalıkları çıkartıp.

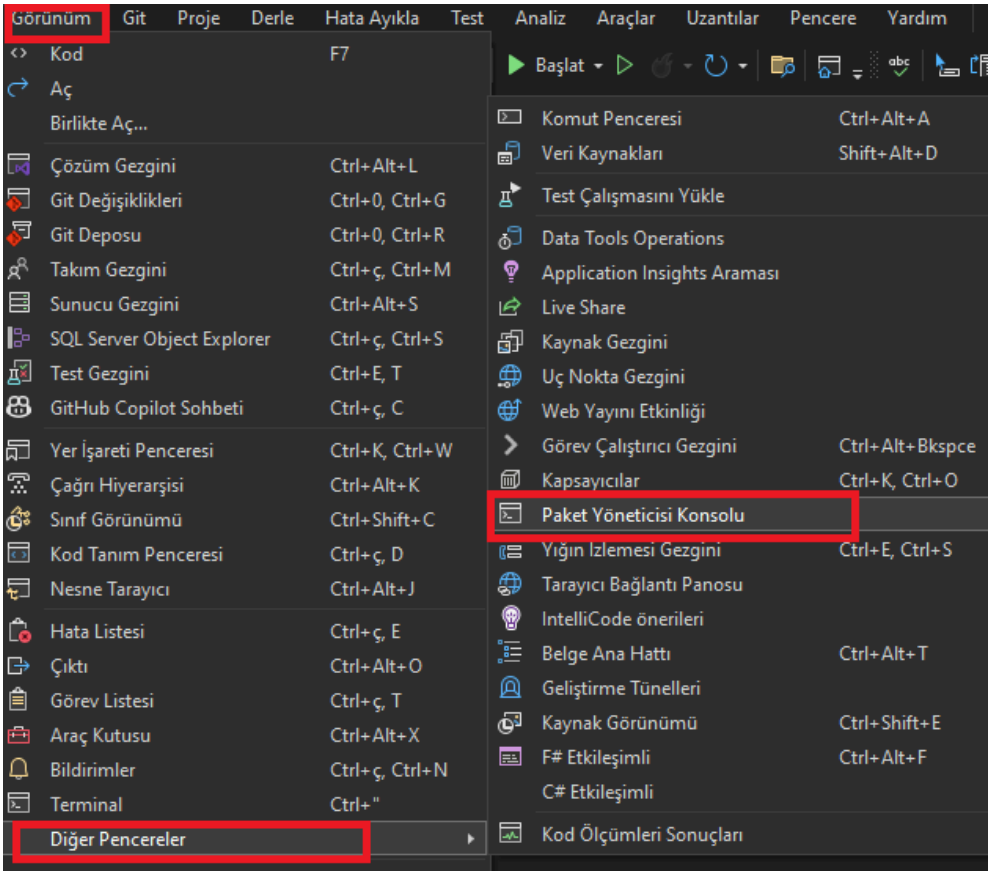
- Daha sonra Business de Concreat klasörümü oluşturop Her nesne için Manager sınıflar açıyorum. Buraya da GenericRepository kısımlarını ekliyorum.



Aşağıdaki gibi diğerlerine de uyguluyoruz.

```
0 başvuru
public class CategoryManager : ICategoryService
{
    ICategoryDal _categoryDal;
    1 başvuru
    public void CategoryDelete(Category c)
    {
        _categoryDal.Delete(c);
    }

    1 başvuru
    public void CategoryInsert(Category c)
    {
        _categoryDal.Insert(c);
    }

    1 başvuru
    public List<Category> List()
    {
        return _categoryDal.List();
    }

    1 başvuru
    public void CategoryUpdate(Category c)
    {
        _categoryDal.Update(c);
    }
}
```

```
0 başvuru
internal class OrderManager : IOrderService
{
    IOrderDal _orderDal;

    1 başvuru
    public Order GetbyId(int id)
    {
        return _orderDal.Get(x => x.OrderID == id);
    }

    1 başvuru
    public List<Order> List()
    {
        return _orderDal.List();
    }

    1 başvuru
    public void OrderDelete(Order o)
    {
        _orderDal.Delete(o);

    }

    1 başvuru
    public void OrderInsert(Order o)
    {
        _orderDal.Insert(o);
    }

    1 başvuru
    public void OrderUpdate(Order o)
```



7.ADIM
TERMİNAL

KONSOLDA
enable-migration

```
PM> enable-migrations
Checking if the context targets an existing database...
PM>
```

enable-migrations yazdığımızda hata almıyorsak şuana kadar doğru yapmışız demektir.

Şimdi Projemize veri tabanımızın adresini vermeliyiz

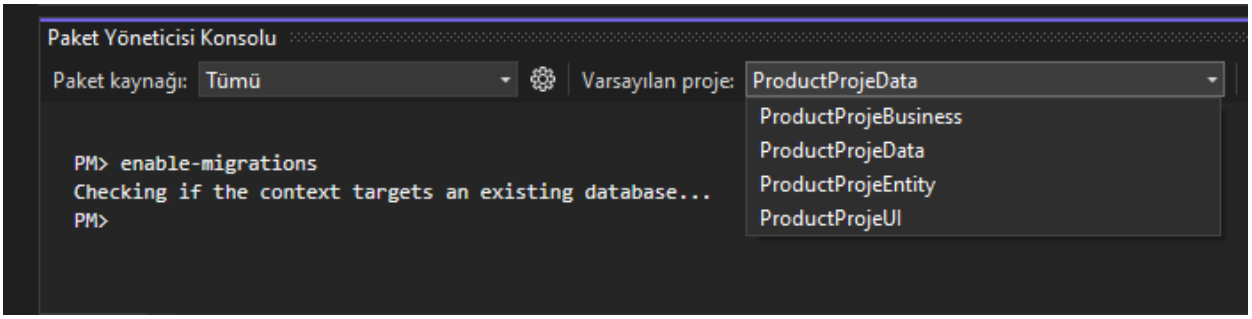
8.ADIM WEB CONFİGE VERİ TABANI BİLGİLERİMİZİ İLETİYORUZ

</runtime> ın altına ConnectionString tag ı açıyoruz.
https://www.connectionstrings.com/sql-server/ sitesinden **Trusted Connection** kopyalayıp yapıştırabiliriz. veritabanımıza girip server, database isimlerini alıp düzeltiyoruz.

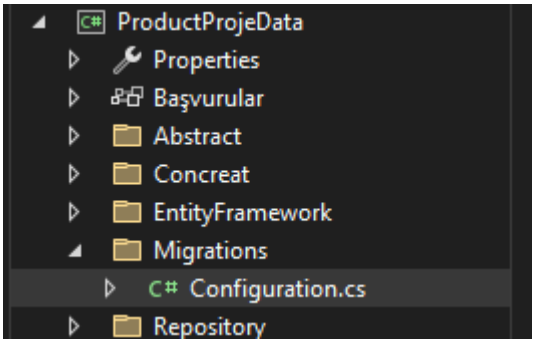
```
    <connectionStrings><add name="Server"
connectionString="Server=AYDEPE\SQLEXPRESS;Database=ProductProje;Trusted_Connection=True;"
providerName="System.Data.SqlClient" /></connectionStrings>
```

veritabanın ismine ProductProje verdik

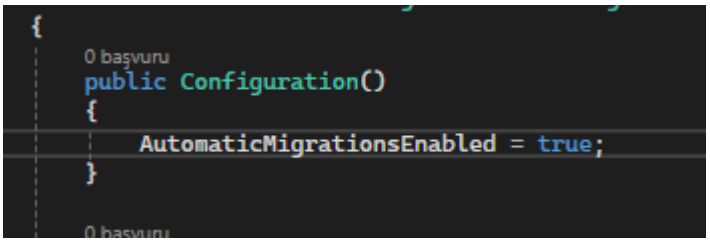
9.ADIM
Veri tabanı bilgilerimizi verdikten sonra migrations klasörümüzü silip tekrar Nuget (paket yöneticisi konsolu)nu açalım
Bu konsolde işlem yaparken Varsayılan projenin DATA olduğuna dikkat edelim.



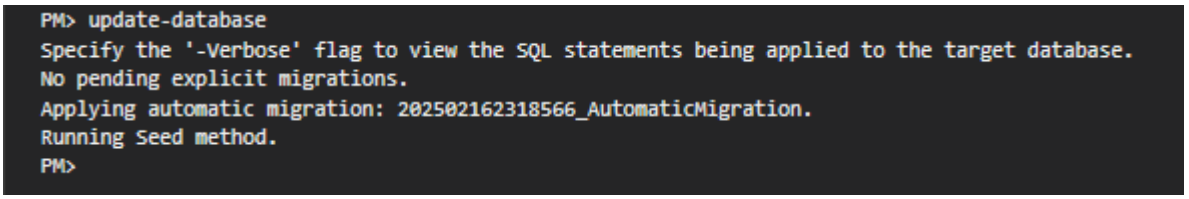
enable-migrations
migrations klasörümüz oluştu



Dosyadaki AutomaticMigrations u true yapıyoruz

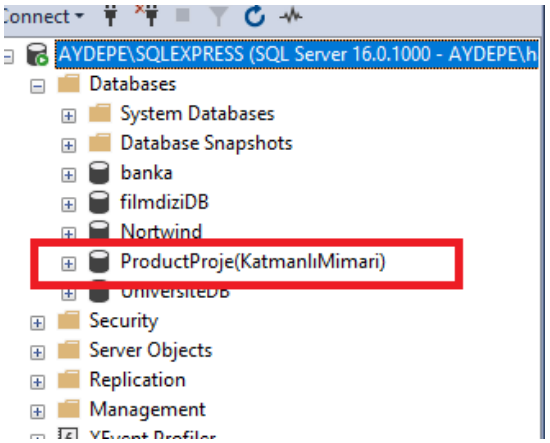


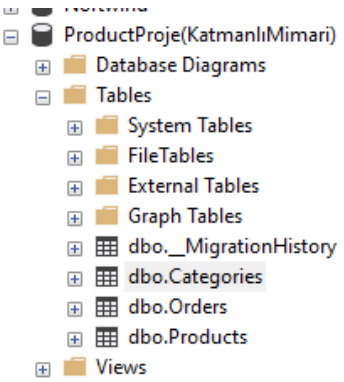
- tekrar konsola “update -database” yazıyoruz



-

böyleda veritabanı oluştu kontrol etmek için MSSQL açıyoruz.





10.ADIM UI KISMINA GEÇİYORUZ..