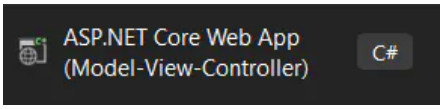
Core Repository

Projeyi .Net Core MVC olarak açıyoruz

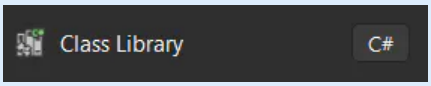


Proje de 3 katman kullanacağız

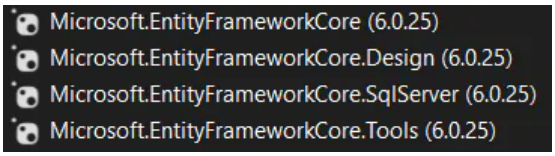
1.UI katmanı

2.Model katmanı tablo verilerin bulundugu katman

3.Data katmanı burada ise verileri veritabanından çekip işlemleri yapıyoruz

> Katmanları oluştururken kullanacagımız proje türü **class library** olacak.
>
> 
>
>
> Eklemek için ise solution üzerine gelerek add diyoruz ve ekliyoruz.

İlk olarak katmanlarımızın referans kısımlarına eklentileri eklememiz gerekecek bunlar:



şekillindedir.

Sonra ise MODEL katmanımıza gelerek tablolarımız için ayarlamalı yaparız. Bunun için katman üzerine gelerek ADD diyerek her bir tablo için yeni bir CLASS oluştururuz ve içerisine tablonun özelliklerini belirtiriz.

```
public class Siparis
{
    [Key]
    0 references
    public int siparisID { get; set; }
    0 references
    public int musteriID { get; set; }
    [ForeignKey("musteriID")]
    0 references
    public Musteri Musteri { get; set; }
    0 references
    public int odemeYontemID { get; set; }
    [ForeignKey("odemeYontemID")]
    0 references
    public OdemeYontem OdemeYontem { get; set; }
    0 references
    public int kargoSirketID { get; set; }
    [ForeignKey("kargoSirketID")]
    0 references
    public KargoSirketleri KargoSirketleri { get; set; }
    0 references
    public int urunID { get; set; }
    [ForeignKey("urunID")]
    0 references
    public Urun Urun { get; set; }
    0 references
    public int miktar { get; set; }
    0 references
    public double fiyat { get; set; }
    0 references
    public int siparisDurumID { get; set; }
    [ForeignKey("siparisDurumID")]
    0 references
    public siparisDurum SiparisDurum { get; set; }

}
```

gibi.

Tabloların classları oluşturuldıysa bir sonraki aşamaya geçebiliriz.

Öncesinde katmanların arasında referans işlemlerini gerçekleştir.

Sonra ise Data katmanına gel ve ApplicationDbContext adında bir dosya oluşturup içerisine MODEL katmanınındaki tabloların alanlarını yazıyoruz yani :

```
public class ApplicationDbContext:DbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>options):base(options)
    {

    }
    0 references
    public DbSet<Urun> Uruns { get; set; }
    0 references
    public DbSet<Siparis> Siparis { get; set; }
    0 references
    public DbSet<siparisDurum> SiparisDurums { get; set; }
    0 references
    public DbSet<OdemeYontem> OdemeYontems { get; set; }
    0 references
    public DbSet<Kategori> Kategoris { get; set; }
    0 references
    public DbSet<Adresler> Adreslers { get; set; }
    0 references
    public DbSet<Musteri> Musteris { get; set; }
    0 references
    public DbSet<KargoSirketleri> KargoSirketleris { get; set; }
}
```

Bu aşamada UI katmanına gidip veritabanı baglantılarımızı oluşturucaz bunun için önce appsettings.json alanına gidiyoruz ve server bilgilerimizi giriyoruz

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": { "dbcon": "server=.;Database=odevCoreUOF;Integrated Security=true;" }
}
```

şekilinde.

Sonra ise bu işlemleri PROGRAM.CS de belirtmemiz gerek:

```
var connectionString = builder.Configuration.GetConnectionString("dbcon");//bu alanda kullancagı
veri tabanı yolunu belirtiyoruz.
builder.Services.AddDbContext<ApplicationDbContext>(option =>
option.UseSqlServer(connectionString)); //veritabanına gönderilecek verileri belirtiyoruz.
```

Artık veri tabanınımız oluştura biliriz. İlk olarak **tool>nuget-packge-manager>consol**. Kısımına gelerek **add-migaration** isim diyerek migration oluşturuyoruz ve ardından ise **update-database** diyerek verileri veritabanına gönderiyoruz.

Şimdi ise asıl mevzuya geliyoruz:

Data katmanına repositori işlemleri için bir klasör açıyorum işlemler düzenli olsun diye.

Klasör içine bir **interface** açıyorum, bu interface tüm yapılacak olan işlemlerin metotlarını yazıyoruz.

```
public interface IRepository<T> where T : class
{
    0 references
    void Add(T entity);
    0 references
    void Update(T entity);
    0 references
    void Remove(T entity);
    0 references
    T getById(Expression<Func<T, bool>> filter, string? includeProperties = null);
    0 references
    IEnumerable<T> GetAll(Expression<Func<T, bool>>? filter = null, string? includeProperties = null);
    0 references
    void removeRange(IEnumerable<T> entities);

}
```

> Alanları tanımlarken **public** olarak tanımlamayı unutmayın.

Şimdi ise bu alanları yazdıktan sonra her tabloya özel **interface** tanımlayalım. Bunun için her tablo için bir **interface** oluşturup metotları yazdıgımız **interfaceden** kalıtım alıyoruz.

```
public interface ISiparis:IRepository<Siparis>
{

}
```

gibi.

Şimdi ise sıra bu metotları doldurmaya geldi bunun için yeni bir class oluşturup metotları implament edicez..

```
public readonly ApplicationDbContext _context;
        internal DbSet<T> _dbset;
        public Repository(ApplicationDbContext context)
        {
            _context = context;
            _dbset = _context.Set<T>();
        }
```

Diyerek veritabanı ile projedeki modelleri eşliyoruz. Metotların için dolduralım şimdi.

```
public void Add(T entity)
        {
            _dbset.Add(entity);
        }
//Add kodu: Veritabınana veri ekleme kodu
public void Remove(T entity)
        {
            _dbset.Remove(entity);
        }
//Remove kodu: Veritabanından veri silme kodu
        public void RemoveRange(IEnumerable<T> entities)
        {
           _dbset.RemoveRange(entities);
```

```
        }
//RemoveRange: Veritabnından baglantılı tabloların
        public void Update(T entity)
        {
            _dbset.Update(entity);
        }
//Update: Veritabanı verileri güncelleme kodu
 public T GetFirstOrDefault(Expression<Func<T, bool>> filter, string? includeProperties = null)
        {
            IQueryable<T> query = _dbset;//IQueryable, LINQ sorgularını değerlendirmek için
kullanılan bir türdür
            if (filter != null)//eger filtreleme sonucunda deger gelirse sorgu çalışır
            {
                query = query.Where(filter);
            }
            if (includeProperties != null)//ilişkili tablo varsa ilişkili tabloları
getirir(isimleri yazar)
            {
                //resimleri getirmek için kullandıgımız alan.
                //split ise ayırma işlemi yapan string fonksiyon
                foreach (var item in includeProperties.Split(new char[] { ',' },
StringSplitOptions.RemoveEmptyEntries))
                {
                    query = query.Include(item);
                }
            }
            return query.FirstOrDefault();
        }
//GetFirstOrDefault: id ye göre verileri getirmede kullanılır genel olarak.
public IEnumerable<T> GetAll(Expression<Func<T, bool>> ? filter=null, string? includeProperties
= null)
        {
            IQueryable<T> query = _dbset;//IQueryable, LINQ sorgularını değerlendirmek için
kullanılan bir türdür
            if (filter!=null)//eger filtreleme sonucunda deger gelirse sorgu çalışır
            {
                query = query.Where(filter);
            }
            if (includeProperties !=null)//ilişkili tablo varsa ilişkili tabloları
getirir(isimleri yazar)
            {
                //resimleri getirmek için kullandıgımız alan.
                //split iseayırma işlemi yapan string fonksiyon
                foreach (var item in includeProperties.Split(new char[]
{','},StringSplitOptions.RemoveEmptyEntries))
                {
                    query=query.Include(item);
                }
            }
            return query.ToList();
        }
```

Şimdi ise Data katmanının içindeki Repository klasörün içine her tabloya özel yazılan metotları temsil etmesi için kalıtım bırakacaz.

```
public class SiparisRepository:Repository<Siparis>,ISiparis
{
    private ApplicationDbContext _context;
    0 references
    public SiparisRepository(ApplicationDbContext context) : base(context)
    {
        _context = context;
    }
}
```

gibi. Burada Repository<Siparis> diyerek Repositorydeki T'nin yerine siparis modelinin gelecegini belirtiyoruz.

```
public class Repository<T> : IRepository<T> where T : class
    {
        public readonly ApplicationDbContext _context;
        internal DbSet<T> _dbset;
        public Repository(ApplicationDbContext context)
        {
            _context = context;
            _dbset=_context.Set<T>();
        }
        public void Add(T entity)
        {
            _dbset.Add(entity);
        }

        public IEnumerable<T> GetAll(Expression<Func<T, bool>> ? filter=null, string?
includeProperties = null)
        {
            IQueryable<T> query = _dbset;
            if (filter!=null)
            {
                query = query.Where(filter);
            }
            if (includeProperties !=null)
            {
                //resimleri getirmek için kullandıgımız alan.
                //split iseayırma işlemi yapan string fonksiyon
                foreach (var item in includeProperties.Split(new char[]
{','},StringSplitOptions.RemoveEmptyEntries))
                {
                    query=query.Include(item);
                }
            }
            return query.ToList();
        }

        public T GetFirstOrDefault(Expression<Func<T, bool>> filter, string? includeProperties =
null)
        {
            IQueryable<T> query = _dbset;
```

```
            if (filter != null)
            {
                query = query.Where(filter);
            }
            if (includeProperties != null)
            {
                //resimleri getirmek için kullandıgımız alan.
                //split iseayırma işlemi yapan string fonksiyon
                foreach (var item in includeProperties.Split(new char[] { ',' },
StringSplitOptions.RemoveEmptyEntries))
                {
                    query = query.Include(item);
                }
            }
            return query.FirstOrDefault();
        }

        public void Remove(T entity)
        {
            _dbset.Remove(entity);
        }

        public void RemoveRange(IEnumerable<T> entities)
        {
           _dbset.RemoveRange(entities);
        }

        public void Update(T entity)
        {
            _dbset.Update(entity);
        }
    }
```

ISiparis diyerek ise o interface deki metotları almamızı sağlar.

> Neden Isiparis interfaceini kalıtım bırakıyoruz.
>
> Eger ki siparişe özel bir metot yazmak istersek gidip **Isipariş alanına yazıcaz böylelikle diger tablolar etkilenmeyecek** ve sadece **siparisRepository** alanına gelecek **Isiparis** alanındaki **metot**.

Her alana entegre ettikten sonra içelerine ana sınıftaki veritabanı baglantısı ise yeni classtaki veri tabanı baglantısını eşlmek gerekir:

```
public class SiparisRepository:Repository<Siparis>,ISiparis
    {
        private ApplicationDbContext _context;
        public SiparisRepository(ApplicationDbContext context) : base(context)//base sözcügü ana
class'ı ifade eder.
        {
            _context = context;//
        }
    }
```

DI(Dependency injection): Katmanlı mimariler üzerinde projeler geliştiriyorsanız, üst seviyede kullanılan classların, alt seviye classlara bağımlı olmamasının sağlanması gibi düşünebiliriz. Projenin yükünün azaltılması ve bağımlılıkların en aza indirgenmesi olarakta tanımlayabiliriz.

Yani nesneler arası bağlantı kurarak kodlamayı azaltırız ve aynı zamanda sistemin güvenirliğini artırırız.

Tüm işlemler bitti ise sıra geldi Program.cs içine DI(Dependency injection) eklemeye bunun için addScoped özelligi kullanacagız.

```
//Bu kodta (builder.Services) demek DI kullanıyoruz anlamına gelir.
//Addscoped yöntemini kullanıyoruz burada. AddTransient  ve AddSingleton  gibi yöntemleri
mevcut.
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));//generic oldugu için
typeof dedik
builder.Services.AddScoped<IUnityOfWork, UnityOfWork>();//generic olmadıgı için gerek olmadı
```

Şuana kadar yapılan işlemler ile artık bir **controller** oluşturup içini doldurup bir **view** sayfası oluşturabilirsiz. Ama biz sitemizin bir işlem sırasında oluşabilecek sorun sirasında işlemi iptal eden sistem için **Unityofwork** kullanıcaz.

İlk önce bir Data katmanı içerisine interface açıyoruz.

```
public interface IUnityOfWork:IDisposable//IDisposable : Aslında biz referans degerli
değişkenlerimizi nesnlerimizin using ile bellekten temizliyorduk. Bunu yapan interface ise
IDisposable bu görevi verir.
    {
        ISiparis Siparis { get; }
        IKategory Kategory { get; }
        IMusteri Musteri { get; }
        IKargoSirket KargoSirket { get; }
        IAdres Adres { get; }
        IOdemeYontem OdemeYontem { get; }
        ISiparisDurum SiparisDurum { get; }
        void save();
        //burada ise InterfaceTablolarımız ile içerisini okuyoruz.
    }
```

Sonra ise buradaki metotların içlerini doldurmak için bir class açıp metotları implament ediyoruz.

Burada artık tüm görevleri UnityOfWork class'ı üstlenmiş oldu. Yapacagımız tüm işlemleri bunu çagırarak yapacagız.

```
public class UnityOfWork : IUnityOfWork//kalıtım bıraktık
    {
        private readonly ApplicationDbContext _context;
        public UnityOfWork(ApplicationDbContext context)
        {
            _context = context; //veritabanı baglantılarını oluşturduk
        }
```

```
        public void Dispose()
        {
            _context.Dispose();
        }
        public void save()
        {
            _context.SaveChanges();
        }
        //programdaki tablolar ile veritabanındaki tabloları eşeltirdik
        public ISiparis Siparis => new SiparisRepository(_context);

        public IKategory Kategory => new KategoryRepository(_context);

        public IMusteri Musteri => new MusteriRepository(_context);

        public IKargoSirket KargoSirket => new KargoSirketRepository(_context);

        public IAdres Adres => new AdresRepository(_context);

        public IOdemeYontem OdemeYontem => new OdemeYontemRepository(_context);

        public ISiparisDurum SiparisDurum => new SiparisDurumRepository(_context);
    }
```

UNİTYOFWORK işlemi 'de bu kadar. Model ve Data katmanı işlemleri bitti.

UI katmanı işlemleri başladı.

Öncelikle katmanı farklı bir şekilde kullanacağız. Bunun için katmanın üzerine tıklayıp New Scaffolding diyoruz ve ekliyoruz.

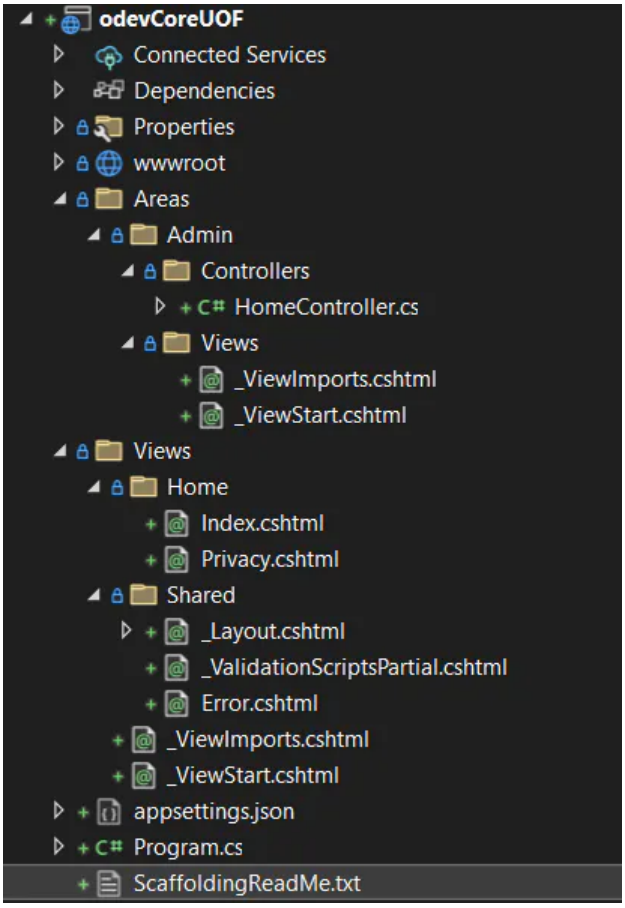
ilk olarak program.cs ye giderek başlangıç noktasını değiştiriyoruz

```
app.MapControllerRoute(
    name: "default",
    pattern: "{area=Admin}/{controller=Siparis}/{action=Index}/{id?}");
```

olarak.

Gereksiz alanları sil (Data, Model) gibi, illa silmek zorunda değilsin ama ortam daha rahat olsun diye. Son olması gereken:

> ⚠️ **ErrorView dosyasını Model katmanına atın silmeyin**



Evet şimdi ise asıl olaya gelebiliriz. Şimdi **controllerleri** ve **viewleri** kodlayacagız.

Areas içindeki **controllere** gel ve yeni controller ekle.

Controllere geldiginde ilk yapman gereken şey önce hangi Area baglı oldugunu belirtmek olmalıdır.

```
[Area("Admin")]
    public class KategoriController : Controller
    {

        private readonly IUnityOfWork _unityOfWork;
        public KategoriController(IUnityOfWork unityOfWork)
        {
            _unityOfWork = unityOfWork;
        }
        public IActionResult Index()
        {
            IEnumerable<Kategori> kategoriList = _unityOfWork.Kategori.GetAll();
            return View(kategoriList);
        }
        public IActionResult Create()
        {
            return View();
        }
        [HttpPost]
        public IActionResult Create(Kategori kategori)
        {
```

```
            _unityOfWork.Kategori.Add(kategori);
            _unityOfWork.save();
            return RedirectToAction("Index");
        }
        public IActionResult Edit(int? id)
        {
            if (id == null || id <= 0)
            {
                return NotFound();
            }
            var kategori = _unityOfWork.Kategori.getById(k => k.kategoriID == id);
            return View(kategori);
        }
        [HttpPost]
        public IActionResult Edit(Kategori kategori)
        {
            _unityOfWork.Kategori.Update(kategori);
            _unityOfWork.save();
            return RedirectToAction("Index");
        }
        public IActionResult Delete(int id)
        {
            var kategori = _unityOfWork.Kategori.getById(k => k.kategoriID == id);
            _unityOfWork.Kategori.Remove(kategori);
            _unityOfWork.save();
            return RedirectToAction("Index");
        }
    }
```

gibi kodlamaları yapabilirsiniz.

İlişkili tablolar için yapmanız gereken bir kaç işlem var.

Class da yapılması gereken:

Burada kullanıcının ilişkili alanları bir liste şekillinde getirmesini sağlar ve aynı zamanda geçerli tablonun alanlarını almamızı sağlar.

```
namespace ModelLayer.VM
{
    6 references
    public class SiparisVM
    {
        15 references
        public Siparis Siparis { get; set; }
        2 references
        public IEnumerable<SelectListItem> MusteriList { get; set; }
        2 references
        public IEnumerable<SelectListItem> OdemeYontemList { get; set; }
        2 references
        public IEnumerable<SelectListItem> KargoSirketList { get; set; }
        2 references
        public IEnumerable<SelectListItem> SiparisDurumList { get; set; }
        3 references
        public IEnumerable<SelectListItem> UrunList { get; set; }
    }
}
```

Controller tarafına yazılması gereken:

Buradan ise kullanıcının view sayfasına verilerin aktarılmasını sağlıyoruz.

```
SiparisVM siparisVM = new SiparisVM()
{
    Siparis = new Siparis(),
    MusteriList = _unityOfWork.Musteri.GetAll().Select(m => new SelectListItem
    {
        Text = m.ad,
        Value = m.musteriID.ToString()
    }),
    OdemeYontemList = _unityOfWork.OdemeYontem.GetAll().Select(a => new SelectListItem
    {
        Text = a.ad,
        Value = a.odemeYontemID.ToString()
    }),
    KargoSirketList = _unityOfWork.KargoSirket.GetAll().Select(a => new SelectListItem
    {
        Text = a.ad,
        Value = a.kargoSirketID.ToString()
    }),
    SiparisDurumList = _unityOfWork.SiparisDurum.GetAll().Select(a => new SelectListItem
    {
        Text = a.ad,
        Value = a.siparisDurumID.ToString()
    }),
    UrunList = _unityOfWork.Urun.GetAll().Select(a => new SelectListItem
    {
        Text = a.ad,
        Value = a.urunID.ToString()
    })
};
```

View tarafı:

Burada ise kullanıcın ilgili tablodaki verileri görebilmesi için ilgili tablonun controller kısımındaki text degerine verilen degeri seçicez.

```
@foreach (var item in Model)
{
    <tr>
        <th>@item.siparisID</th>
        <th>@item.Musteri.ad</th>
        <th>@item.OdemeYontem.ad</th>
        <th>@item.KargoSirketleri.ad</th>
        <th>@item.Urun.ad</th>
        <td>@item.miktar</td>
        <td>@item.fiyat</td>
        <td>@item.SiparisDurum.ad</td>
        <td>
            <a asp-controller="Siparis" asp-action="Crup" asp-route-id="@item.siparisID" class="btn btn-outline-warning">Düzenle</a>
            || <a asp-controller="Siparis" asp-action="Delete" asp-route-id="@item.siparisID" class="btn btn-outline-danger">Sil</a>
        </td>
    </tr>
}
```