

# Asp.Net Core N-Tier Architecture Giriş dersi ders notları (my yazılım)

Bu notlar daha sonra deftere geçirilmek üzere ders kayıtlarını izlerken hazırladığım notlardır.

N-tier Architecture => N katmalı / çok katmanlı mimari 4 adet Katman görcez

1. **Entity Layer** olacak bu Entity Layer içerisinde yapılarımız olacak (entity'ler bizim sql e yansıtacağımız yani sql deki tablolarımıza karşılık gelen)
2. **Data Access layer** => Veri tabanı işlemleri ve bağlantı adreslerini tutacağımız yerdir. (Veri erişim katmanı)
  - Context sınıfımız bulunacak,
  - veri tabanı bağlantı adresimiz contextin içerisinde yer alacak, ve içerisinde DbSet türünde (mvc deki gibi) o yapıyı data access'in içerisinde tutcaz,
  - burada bir de Desing pattern kullanılacak o da Repository olacak yani repository design pattern (Repository Design Patter ise bize her seferinde bütün controllerlarda her seferinde contextten newleme yapmayacağız (hatırlarsan ... context = new Context tarzı bir şey yapardık ona gerek kalmıcaz)
  - Repository design pattern içerisinde oluşturacağımız Generic yapı sayesinde tüm controllerlarımızda Context sınıfımızı çağırmadan işlemlerimizi yapabiliriz.
3. **Business Layer Katmanı** => İş Kuralları
  - Yani mesela Business içerisinde servisler yazacağız, (servisler içerisinde mesela parametreden gelen değer == bool 'sa sunu yap else => ise bunu yap gibi yada validasyon işlemleri gibi ya da parametreden gelen değer en fazla şu kadar karakter olabilir gibi koşul ifadeleri yer alacak bunlar bizim iş kurallarımız olacak (projenin içerisinde geçerli koşullar)
4. **Presentation Layer (UI Katmanı)** => Projenin ön yüzünü oluşturan Model – View - Controller yapılarının yer aldığı (MVC) katmandır.

## SOLID PRENSİBİ =>

**S : Single Responsibility Principle** => Her oluşturduğumuz Sınıf(Nesne farklı bir işlem yapacak, yalnızca o sınıf içerisinde geçerli olan)

**O: Open Closed Principle** => İçerisindeki metotlara erişebilir olacak ama değiştirilmesine izin verilmeyecek

**L: Liskov Substitution Principle** => Kodlarda herhangi bir değişiklik yapmadan alt sınıfları türedikleri (üst) sınıflar tarafından kullanabilmeliyiz. Mesela bazı sınıflara yanına " : " yazıp başka bir sınıftan miras aldırıyoruz. Bu sayede o sınıftaki metotlara da diğer sınıflar erişebiliyor. Böylece içerisinde değişiklik yapmadan o sınıftan başka bir sınıf türetebiliyoruz.

**I: Interface Segregation Principle** => Arayüz

- Bütün metotları sınıflarda tutuyoruz ama sınıfların içerisinde çağırıyoruz controller tarafında.
- Burada Interface kavramı devreye giriyor.
- Interface içerisinde metodun dönüş tipini, adını parametrelerini yazıyoruz ama o metodun ne iş yapacağını sınıfta tanımlarız yani interface'ten miras alan bir sınıf olacak.

**D: Dependency Inversion Principle** => Sınıflar arası bağımlılıklar olabildiğince az olmalıdır. Özellikle Üst sınıflar alt sınıflara bağımlı olmamalıdır.

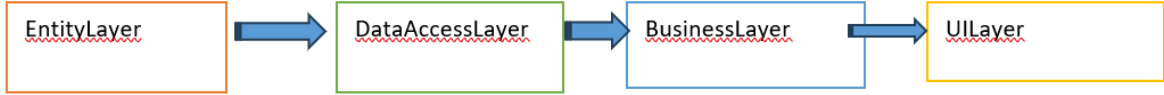
Solid prensiplerine uygun yapmalıyız projeyi.

## Proje Geliştirme Aşaması

İlk Asp.Net Core tabanlı projemizin kurulum, katmanları oluşturma aşamalarını not edelim. Bundan sonraki proje oluşturma işlemleri de benzer olacaktır. Bu notlar bana bir taslak olacak.

1. İlk olarak Create New Project diyip **Boş bir Blank Solution** oluşturduk adına **MyAcademyOneMusic** dedik. Bu solution'un içerisine katmanlarımızı oluşturcaz.
2. Solution'a sağ tık yapıp Add => New Project
3. Arama kısmına Class Library => Yanında C# simgesi olan Class Library Seçilir.
4. Solutionumuzun adı **MyAcademyOneMusic** 'ti, Projemizin adına **OneMusic.EntityLayer** (Entity Katmanı : içerisinde classlarımızı / tablo oluşturcaz)
5. .Net 8.0 çıkacak otomatik olarak, next deriz.
6. **OneMusic.EntityLayer** katmanımız oluştu içerisinde boş bir class oluşturcaz onu siliyoruz. Sadece Dependency kalcak
7. Solutiona tekrar sağ tık yapıp Ad => New Project deriz. Tekrar bir ClassLibrary seçeriz.
8. **OneMusic.DataAccessLayer** adını veriyoruz bu sefer ve oluşturuyoruz. İçindeki class ı yine siliyoruz.
9. Ardından da yine aynı işlemleri yaparak **OneMusic.BusinessLayer** Katmanını oluşturuyoruz.
10. En son ise tekrar yeni proje oluşturup bu sefer Asp.Net Core Web App (Model-View-Controller) olan seçeneği (yanında C# yazan) seçeneği seçiyor ve adına **OneMusic.WebUI** diyebiliriz (PresentationLayer ' da denebilir), ardından next deriz ve proje oluşturulur.
11. Asp.Net Core Web App (Model-View-Controller) => bu yapı bize hazır mvc yapısını oluşturuyor , model controller view vs.

- Entity Katmanımızı DataAccess Katmanına Referans vercez (proje referansı olarak). Entity katmanındaki verilerimizi (classlar) DataAccess Katmanında kullanabilmemiz için referans vermemiz gerekir.
- DataAccess Katmanımızı da Business Katmanımıza referans vercez
- Business katmanımızı da UI Katmanına referans vercez
- Böylelikle UI katmanı tüm katmanlara erişebilir olacak



Projemizde Katmanları referans verme işlemini şu şekilde yapıyoruz:

1. İlk olarak Entity Katmanımızı DataAccess Katmanına Referans vercez. Bunun için DataAccess Katmanında Dependency'nin üzerine sağ tıklayıp Add Project Reference 'ı tıklıyoruz ve çıkan katmanlar arasından EntityLayer katmanını seçiyoruz. Böylelikle DataAccess katmanında Dependency içerisine Project bölümü geliyor ve içerisinde OneMusic.EntityLayer'i görebiliyoruz. (başarıyla reference edilmiş oluyor)
2. Şimdi de aynı işlemi BusinessLayer için yapıyoruz. DataAccess Katmanımızı da Business Katmanımıza referans vercez. Bunun için Business Katmanında Dependency'nin üzerine sağ tıklayıp Add Project Reference 'ı tıklıyoruz ve çıkan katmanlar arasından DataAccessLayer katmanını seçiyoruz. Böylelikle BusinessLayer katmanında Dependency içerisine Project bölümü geliyor ve içerisinde OneMusic.DataAccessLayer'i görebiliyoruz. (başarıyla reference edilmiş oluyor)
3. Şimdi de aynı işlemi WebUI için yapıyoruz. Business Katmanımızı da WebUI Katmanımıza referans vercez. Bunun için WebUI Katmanında Dependency'nin üzerine sağ tıklayıp Add Project Reference 'ı tıklıyoruz ve çıkan katmanlar arasından BusinessLayer katmanını seçiyoruz. Böylelikle WebUI katmanında Dependency içerisine Project bölümü geliyor ve içerisinde OneMusic.BusinessLayer'i görebiliyoruz. (başarıyla reference edilmiş oluyor)
4. Böylelikle UI katmanı tüm katmanlara; Business katmanı DataAccess dolayısıyla Entity katmanına; DataAccess katmanı ise Entity katmanına erişebiliyor olur.

Şimdi artık kodlarımızı yazmak için dosya ve klasörlerimizi oluşturmaya başlıyoruz:

İlk olarak Classları oluşturmamız gerek. Yani tablolarımızı. Dolayısıyla önce EntityLayer katmanına Sağ tık yapıp Add New Folder deriz ve "Entities" adında boş bir klasör oluştururuz. Bunun içerisinde add => class diyerek classlarımızı oluşturmaya başlayabiliriz.

```
namespace OneMusic.EntityLayer.Entities
{
    public class Banner
    {
        public int BannerId { get; set; }
        public string Title { get; set; }
        public string SubTitle { get; set; }
        public string ImageUrl { get; set; }
    }
}
```

Şuradaki örnek bir class örneğini inceleyelim;

Classlarımızı ilk oluşturduğumuzda bizlere internal olarak oluşturur. Yani internal class Banner {} olarak. Biz bunu “public” yaparız. Sebebi ise Diğer katmanlardan erişebilir olması içindir. Eğer internal olarak bıraksaydık o sınıfa başka katmanlardan erişemezdik.

### NOT:

**Internal:** sadece bulunduğu katmandan erişilebilir( bulunduğu katman EntityLayer içerisindeki her yerden erişilebilir ama DataAccesssten erişilemez.

**Public:** tüm katmanlardan erişilebilir. \*\*\* en çok kullanılan

**Private:** Sadece bulunduğu sınıftan erişilebilir.

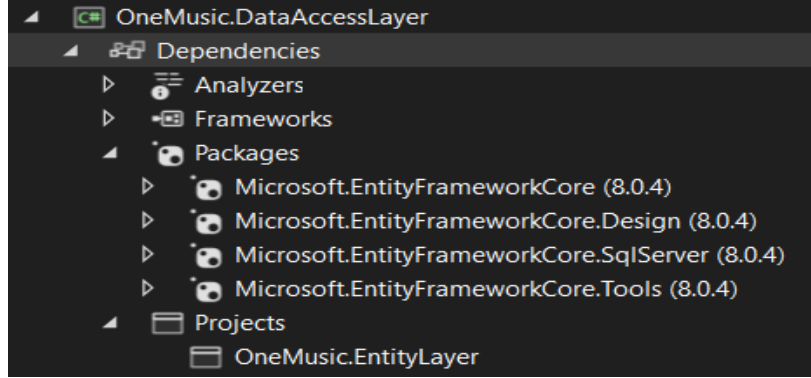
**Protected:** hem bulunduğu sınıf içerisinde, o sınıftan miras alan (alt sınıflar) sınıflardan erişilebilir.

Classlarımızı oluşturduktan sonra DataAccess katmanına gelip sağ tıklayınız ve yeni bir klasör oluştururuz. Bu klasör ise “Context” klasörü olur. Contextlerimizi bu katmanda tutarız. “OneMusicContext” adında bir class oluşturuyoruz. Bağlantı adreslerimizi buraya geçeriz.

Bunu yaptıktan sonra Code First yapısını kullanacağımız için önce paketlerimizi yüklememiz gerek. Burası önemli bir kısım:

- **DataAccess Katmanında Dependency** kısmına sağ tıklayıp **Manage NuGet Packages** Açarız.
- Browse'ta aramalar kısmına entity yazıp çıkan “**Microsoft.EntityFrameworkCore**” paketini seçeriz. 8.0.4 olan
- Daha sonra yine aynı yerde altlarda “**Microsoft.EntityFrameworkCore.Design**” paketini de yükleriz. 8.0.4 olan (şuna dikkat edilmeli; yüklediğimiz her paketin sürümü aynı olsun 8.0.4 olsun gibi)
- Daha sonra yine aynı yerde altlarda “**Microsoft.EntityFrameworkCore.SqlServer**” paketini de yükleriz. 8.0.4 olan
- Son olarak da yine aynı yerde altlarda “**Microsoft.EntityFrameworkCore.Tools**” paketini de yükleriz. 8.0.4 olan

Paketlerimizi yükledikten sonra DataAccessLayer katmanında Dependency içerisine Packages bölümü geliyor ve yüklenen 4 paketlerimizi de oradan görebiliyoruz.



### NOT: Başlangıç katmanını (projesini) seçme;

Şimdi 4 katman oluşturduk ve her birinde işlemler yapacağız. Fakat projeyi çalıştıracağımız katman UI katmanı olduğu için bizim başlangıç projesini **OneMusic.WebUI** olarak seçmemiz gereklidir. Bunun için **OneMusic.WebUI** 'ın üzerinde gelip sağ tıklayıp **"Set as Startup Project"** seçeneğini seçmemiz gereklidir.

Entity Katmanında Classları oluşturduktan sonra; DataAccess katmanında Context sınıfımızı oluşturup (OneMusicContext) bu context içerisinde

- DbSet'ten miras alma =>> `public class OneMusicContext : DbContext`
- DbSet'leri oluşturma (  
`public DbSet<About> Abouts { get; set; }`  
`public DbSet<Album> Albums { get; set; } ..`
- Database ile bağlantı yapabilmek için bağlantı adresini verme  
`optionsBuilder.UseSqlServer("server=CAGLA\\SQLEXPRESS;database=OneMusicDb;integrated security=true;trustServerCertificate=true");`  
gibi işlemleri yaptık.

Sonrasında WebUI katmanında EntityFramework'un Design paketi de olması gerektiği için WebUI Katmanının Dependency bölümüne gelip Manage NuGet Packages 'i açıyoruz:

- **Microsoft.EntityFrameworkCore.Design** paketi yüklendi

1:33:30 da kaldı.

### NOT

#### SQL TARAFINA PROJEYİ DAHİL ETME

Şimdi Package Manager Consolumuzu seçeriz. Ve Default Project kısmını ➔ DataAccess katmanını seçmemiz gerekir.

DataAccess katmanını seçme sebebimiz, Context sınıfımız nerdeyse (bağlantı adresimiz Context sınıfında) o katmanda migration oluşturcaz.

- **PM> add-migration initial** → ((başlangıç migrationu)) bu işlem bize otomatik bir migration oluşturur. Adını initial dememizin sebebi ise (bize kalmış istediğimizi yazabiliriz add-migrationdan sonra) ilk migrationumuz olmasıdır.
- Migrationı oluşturduktan sonra ise **PM> update-database** → deriz (bu işlemi mvc code first yapısında da yapıyorduk entity tarafında her değişiklik yaptığımızda.) bu işlemden sonra sql tarafına tüm tablolarımız ve tablolar arası ilişkiler yansımış oldu.

## Bu adımdan sonra yapılanlar: DataAccess katmanı Abstract klasörü

- DataAccess katmanına gelip yeni bi klasör oluşturduk → **Abstract**
- Abstract klasörü içerisinde **Interface** 'lerimizi oluşturcaz. Bunun için Abstract klasörüne gelip sağ tık yapıp **"New İtem"** 'ı seçeriz ve çıkan ekranda **Interface** seçeneğini seçeriz.
- Interface 'lerde ise IAboutDal, ISingerDal, IContactDal gibi isimlendirme yapmamız gerekir. Yani başına I sonuna ise Dal gelir (Dal - Data Access Layer) Böylelikle interface olduğunu ayırt edebiliriz.

Uygun adlandırma → I + EntityName + Dal

- **NOT:** Oluşturduğumuz class, interface... vs internal olarak ekliyor fakat biz bunu her zaman public olarak değiştireceğiz unutma!!
- Tek tek her bi interfacede ekleme silme güncelleme işlemleri için metot yazmak yerine; bir generic interface oluşturup onda tanımlayacağız ve tüm interface'ler **IGenericDal** interface'inden miras alacaktır. Böylelikle her interface generic interface içerisindeki metotlara erişim sağlayacak. Detaylı Notlar OneMusic Projesinin içindedir.
- ⇒ **public interface IAboutDal : IGenericDal<About> {}** → T yerine About geliyor yani GenericRepository'deki (IGenericDal'dan miras alıyordu) tüm metotlarda T yerine About sınıfı geçiyormuş gibi işlem yapacak, diğer classlar içinde durum böyle. Miras almanın önemi burda belirtilmiştir.

## Bu adımdan sonra yapılanlar: DataAccess katmanı Repositories klasörü

- DataAccess katmanında Repository klasörü açılır,
- GenericRepository classı oluşturulur.
- Metotların imzasını interfacede (IGenericDal) yaptık içerisinde ise (yapacağı işlemler) repository de ( GenericRepository ) doldurcaz.

- `public class GenericRepository<T> : IGenericDal<T> where T : class` → Generic Repository, IGenericDal Interface'inden Miras alır zaten implement edildiğinde IGenericDal içerisindeki tüm metotlar buraya geliyor. (GetList, GetById, Delete, Update, Create gibi)
- ⇒ `private readonly OneMusicContext _context;` → context'imizi burada çağırıyoruz. \_context üzerine gelip ctrl + . ile "generate constructor'ı" seçerek constructor'u oluşturuyoruz.
- ⇒ `public GenericRepository(OneMusicContext context) //oluşturulan constructor`  

```
{
    _context = context;
}
```

**NOT → constructor yapısı :** GenericRepository sınıfı ile işlem yapacağımız zaman bu sınıfı başka bir yerde çağırdığımız anda "OneMusicContext" ile hemen bir nesne (context) örnekler ve bu nesneyi de hemen "\_context = context" ile eşitler

- ⇒ biz MVC yapısında sınıfımızı OneMusicContext context = new OneMusicContext() olarak new'lerdik. Fakat bir sınıfı new'lemek o sınıfa bağımlılık demek oluyor.
- ⇒ biz ise sınıfa bağımlı olmak yerine o sınıftan bir parametre türetip (context) o parametreye de field'i atıyoruz ( \_context ).
- ⇒ Ayrıca zaten Interface'ler de new'lenemez. Biz zaten controllerlarımız içerisinde de işlemlerimizi yaparken sınıflarımızı new'leyerek yapmayacağız. hep interface'leri çağıracağız.
- ⇒ Sınıftan nesne türetmeyeceğiz. Interface'ten bir field ( \_context) türeticez ve o field sayesinde işlemlerimizi yapacağız. Böylelikle hiç bir sınıfa bağımlılığımız olmayacak.

## Bu adımdan sonra yapılanlar: DataAccess katmanı Concrete klasörü

Concrete; interface'lerimizi miras alan sınıflarımızı tutacaktır.

İsimlendirmeleri genel olarak → EfAboutDal, EfAlbumDal.. şeklinde

Uygun adlandırma → Ef + EntityName + Dal

İlk olarak miras alma işlemlerini şöyle yaparız:

```
public class EfAboutDal : GenericRepository<About>, IAboutDal
```

- ⇒ Görüldüğü gibi GenericRepository classından miras alıcak (oradaki metotlar için) ve T yerine ise About entitysi belirtilmiştir.

- ⇒ Ek olarak ,IAboutDal içerisinde farklı bir metot daha yazdırmak istersem yani sadece bu entityye özgü farklı bir metot yazmak istiyorsam (örneğin Count() metodu) IAboutDal a yazabilirim haliyle miras alması için buraya da implement ederiz
- ⇒ constructor : EfAboutDal üstüne gelip Generate constructor'ı dahil ediyoruz

```
public class EfAboutDal : GenericRepository<About>, IAboutDal
{
    public EfAboutDal(OneMusicContext context) : base(context) {}
}
```

- ⇒ **base:** EfAboutDal sınıfı nerden miras almışsa (GenericRepository) bir üst sınıftaki parametreye karşılık geliyor
- ⇒ Entityler(classlar: about,song ...) içerisinde prop dışında başka bir şey yazmamamız daha sağlıklıdır solid prensiplerine göre

Aralarındaki ilişkiyi şöyle özetleyebiliriz:

IAboutDal (miras alır) → IGenericDal

EfAboutDal (miras alır) → GenericRepository + IAboutDal

GenericRepository (miras alır) → IGenericDal

## Bu adımdan sonra yapılanlar: Business katmanı Abstract klasörü

DataAccess Katmanında genel işlemler yapıldı ve artık business katmanına geçebiliriz. Bu katmanda da Abstract ve Concrete klasörlerini oluşturuyoruz.

- BusinessLayer katmanında → Abstract Klasöründe Interface'ler ( IAboutService, IAlbumService.. )
- BusinessLayer katmanında → Concrete Klasöründe Classlar ( AboutManager, AlbumManager.. )

BusinessLayer katmanındaki IGenericService, IAboutService,... mantık olarak DataAccessLayer katmanında mantığa çok benzemektedir.

Gelelim IGenericService Interface'ine:

```
public interface IGenericService<T> where T : class
{
    List<T> TGetList(); //başlarına T ekleriz ki IGenericDal'dan(DataAccessLayer katmanındaki) bunu ayırmak için

    T TGetById(int id);

    void TCreate(T entity);
    void TUpdate(T entity);
    void TDelete(int id);
}
```



---

**Not:** Concrete Klasörlerinde classlar barınır, Bu klasörlerde Generic bulunmaz. Generic İnterfacelerde (Abstract klasörü) oluşturulur. Birde Repositories klasöründe GenericRepository olarak bulunur.

*Classlar İnterfacelerden miras alır ve interfaceleri implement ederek interfacelerin miras aldıkları metotları getirirler.*

*Concrete Klasöründe;*

*EfAboutDal → DataAccess Katmanı*

*AboutManager → Business Katmanı*

---

### Örnek bir AboutManager Classı;

- ⇒ **public class AboutManager : IAboutService** → AboutManager, IAboutService interface'inden miras alıyor. IAboutService üzerine gelip "Implement İnterface diyoruz ve metotlarımızı implement ediyoruz. IAboutService ise IGenericService'ten miras alıyordu haliyle IGenericService içindeki metotlar geliyor.
- ⇒ **private readonly IAboutDal \_aboutDal;** → IAboutDal interface'inden bir \_aboutDal field'i örnekledik. Sınıftan bir nesne örneği türetebilirdik (new'leme) ama interface'ten bir nesne örneği türetemediğimiz için newlenemezdi.
- ⇒ **public AboutManager(IAboutDal aboutDal)** → constructor
  - {
  - \_aboutDal = aboutDal;** → biz sınıfa bağımlı olmak yerine o sınıftan bir parametre türetip (aboutDal) o parametreye de field'i atıyoruz ( \_aboutDal ).
  - }
- ⇒ **public void TCreate(About entity)** → TCreate ise Business katmanında IGenericService 'ten gelen metot
  - {
  - \_aboutDal.Create(entity);** → DataAccessten gelen Create metodu
  - } Bunun gibi metotlar yazılır..

**Bu adımdan sonra yapılanlar: WebUI katmanı Program.cs dosyası**

- ⇒ Programın katmanlar arasında tanımladığımız interfaceri, classları ve metotlarla işlemleri çalıştırabilmesi için her bir katmanda classlar nerden miras almışsa onu burada mutlaka tanımlamamız gerekir.
- ⇒ Unutmayalım ki proje **WebUI** katmanından çalışıyordu ve **Program.cs** de bu katmanın içerisinde.

`builder.Services.AddScoped<IAboutDal, EfAboutDal>();` → **DataAccessLayer**: **IAboutDal** içerisindeki metotlar **EfAboutDal** da yazılmıştır. (registiration işlemi)

`builder.Services.AddScoped<IAboutService, AboutManager>();` → **BusinessLayer**: **IAboutService** içerisindeki metotlar **AboutManager** da yazılmıştır. (registiration işlemi)

`builder.Services.AddScoped<IAlbumDal, EfAlbumDal>();`  
`builder.Services.AddScoped<IAlbumService, AlbumManager>();`

...

`builder.Services.AddDbContext<OneMusicContext>();` → **DbContext**'imiz de **OneMusicContext** olarak belirtiriz. Bunu vermezsek projeyi çalıştıramayız!!

## Bu adımdan sonra yapılanlar: WebUI katmanı Controller oluşturma

MVC deki çok benzer yapıdır. Fakat bazı farklılıklar var onlara değineceğim.

- ⇒ Mesela Conroller oluşturduktan sonra View eklerken → **Razor View** seçilecek. Diğer işlemler (Layouta bağlı olma, create partial ya da hiçbir şeye bağlı olmayan Viewler) MVC dekiyle aynı mantıkta yapılır.

**AdminController** → indexi için eklenecek View hiç birşeye bağlı olmaz

**AdminContactController** → index, addContact, updateContact viewleri, AdminLayout indexine bağlı olur

**Partiallar** → Create a partial olur .. gibi

Controllerlarda farklı olarak metotlarımızda **\_aboutService** şeklinde service field'i çağrılır →

`private readonly IAboutService _aboutService;`

```
public AdminAboutController(IAboutService aboutService) // constructor
{
    _aboutService = aboutService;
}
```

**IAboutService** interface'inden **\_aboutService** field'i türetilir ve constructorda bunu **aboutService** parametresine atarız. Ardından metotlarımızda şu şekilde kullanırız:

```
[HttpGet]
public IActionResult CreateAbout()
{
    return View();
}

[HttpPost]
public IActionResult CreateAbout(About about)
{
    _aboutService.TCreate(about);
    return RedirectToAction("Index");
}
```

MVC de direk contextten bir nesne türetilip (new'leme) o nesne aracılığıyla işlemlerimizi yaparken, artık burada servisler aracılığı ile (Business katmanını → WebUI katmanına referans vermiştik) TCreate, TDelete, TGetList gibi metodlarımızı çağırırız. Controller içerisindeki işlemlerimiz böylece çok daha temiz bir görünüme ulaşır.

---

**Önemli Not:** *Controllerlarda Add View ile view ekleyebilmek ve viewlerde entitylerimizi (Modellerimiz -About, Album .. vs) getirebilmemiz için mutlaka ama mutlaka :*

*\_ViewImports dosyasına → @using OneMusic.EntityLayer.Entities*

*Eklemeyi unutma*

---

Diğer controllerları oluştur, silme işleminde çıkan aleti create ve update içinde yap (başarılı oluşturulmuştur diye. Tüm viewlara yap.  
İconlarımı modernize klasöründe icon-tabler.html den buluyorum.

## Önemli bir Not:

Controller tarafında ilişkili tablolarda eğer iki tablodan veri çekilecekse (yani hem Album hem de onla ilişkili olan Singer tablosu olsun;

```

public class Album
{
    public int AlbumId { get; set; }
    public string AlbumName { get; set; }
    public string CoverImage { get; set; }
    public decimal Price { get; set; }

    public int SingerId { get; set; }
    public Singer Singer { get; set; }
    public List<Song> Songs { get; set; }
}

```

Ben Album tarafında (Indexlerde) DropDownList ile Şarkıcı isimlerini getirmek istiyorsam AlbumControllerda :

```

private readonly IAlbumService _albumService; ➔ _albumService field'ini ve
private readonly ISingerService _singerService; ➔ _singerService field'ini çağırırız

```

```

// Constructor
public AdminAlbumController(IAlbumService albumService, ISingerService singerService)
{
    _albumService = albumService;
    _singerService = singerService;
}

```

➔ Constructor oluştururken de **singerService** parametresini de **\_singerService** field'ine atarız. Yani constructorda onu da eklemiş oluyoruz Album classında kaç tane ilişkili tablo varsa Constructor içerisine hepsinin Business katmanındaki interface'ini, parametresini ve field'ini belirtiriz .

**Önemli bir Not:** ilişkili tablolarda bir tablonun içerisinde başka tabloyu da çağırabilmek

Net Core düzeninde ilişkili tablolarda bir tabloya ait viewda başka bir tabloyu da çağırabilmemiz için daha doğru o tabloya ait herhangi bir sütunu getirebilmemek için DataAccess katmanında yeni bir Metot oluşturmamız gerekebilir. Örnek üzerinden gidelim:

Mesela Album Tablosunda Singer tablosu da vardır yani ilişkilidir bu iki tablo. Ben albüm sayfasında diyelim şarkıcı isimlerini de (Singer tablosundan) getirmek istiyorum. Bunun için

➔ Öncesinde DataAccess katmanındaki EfAlbumDal' a gelirim. Burada:

**public List<Album> GetAlbumsWithSinger()** ➔ bu metot sayesinde album sınıfından gelen propertyler singer sınıfından parametreleri de dahil ederek listelicek.

```
{  
    return _context.Albums.Include(x => x.Singer).ToList(); ➔ album içerisinde singerları da  
    çağırabilmemiz için  
}
```

İşlemine yaparız. Bu işlemle Include() metodunu kullanarak aslında Album sınıfından Singer sınıfını da çağırmış oluyoruz. Ama henüz işlemler bitmedi. Burada oluşturduğumuz **List<Album> GetAlbumsWithSinger()** metodunu IAlbumDal ve Business katmanındaki IAlbumService ve AlbumManager'a da ekleyeceğiz:

**IAlbumDal ➔** **public interface IAlbumDal : IGenericDal<Album>**  
**{**  
 **List<Album> GetAlbumsWithSinger();**  
**}**

**IAlbumService ➔** **public interface IAlbumService : IGenericService<Album>**  
**{**  
 **List<Album> TGetAlbumsWithSinger();**  
**}**

**AlbumManager ➔** **public List<Album> TGetAlbumsWithSinger()**  
**{**  
 **return \_albumDal.GetAlbumsWithSinger();**  
**}**

Bu işlemlerden sonra AlbumConroller'ında Index için:

```
public IActionResult Index()  
{
```

```
var values = _albumService.TGetAlbumsWithSinger();  
return View(values);  
}
```

- ⇒ Görüldüğü gibi **TGetList** değil **TGetAlbumsWithSinger** metodu çağırılır. Haliyle biz Index View'ında **Singer.Name** gibi Singer tablosundan Name'leri getirebiliriz.
- ⇒ Bunu bu şekilde başka ilişkili tablolar için de yapabiliriz.
- ⇒ Zaten yazacağımız yeni bir fonksiyonu biz DataAccess Katmanındaki Classta (EfTableNameDal) tanımlayıp ardından aynı katmandaki Interface ve Business katmanındaki Interface ve sınıfta çağırıyoruz.
- ⇒ Controller tarafında ise o Metodu kullanarak işlemlerimizi yapıyoruz.
- ⇒ Bu durum .Net Core'un Kod temizliği ve düzeni açısından katmanlı mimarinin en önemli özelliklerinden biridir. Kod kirliliğini ortadan kaldırır.

## Önemli bir Not: Validasyon İşlemleri

Validasyon işlemlerimizi yapabilmemiz için işlemlere başlamadan önce ilk olarak fluent validation paketini projemize dahil etmemiz gerekir.

- ⇒ **BusinessLayer** Katmanına geldik. Burada **Dependencies** üzerine sağ tık yapıp **Manage NuGet Packages**'i açıyoruz ve **Browse** kısmına "**fluentvalidation**" yazıyoruz.
- ⇒ **FluentValidation.DependencyInjectionExtensions** paketini yükleriz.
- ⇒ Yükleme işlemi tamamlandıktan sonra **BusinessLayer** katmanındaki **Dependencies** içerisine girip **Packages** bölümünden yüklenip yüklenmediğini kontrol edebiliriz.

## BusinessLayer Katmanında Validators işlemleri:

- ⇒ **BusinessLayer** katmanına sağ tık ile add folder diyip yeni klasör ekleriz. Validasyon işlemlerini bu klasörde yapacağımız için **Validators** adını verebiliriz.

- ⇒ Burada bir class oluşturacağız. Singer Entitymiz için bir validasyon işlemi yapmak istedik diyelim, SingerValidator.cs adında bir class oluşturuyoruz. Ve tüm validasyon komutlarımızı burada ayarlıyoruz. Diğerleri içinde bu şekilde yapıcız
- ⇒ En önemli propertylerimizden birisi **RuleFor**
- ⇒ WithMessage, MaximumLength, MinimumLength .. vs gibi validasyon kontrollerini istediğimiz şekilde burada kullanabiliriz.
- ⇒ Örnek bir validasyon sınıfı:

```
public class SingerValidator : AbstractValidator<Singer>
{
    public SingerValidator()
    {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Şarkıcı Adı Boş Bırakılamaz!").MaximumLength(50).WithMessage("En Fazla 50 Karakter Yazabilirsiniz..").MinimumLength(4).WithMessage("En az 4 karakter yazmalısınız..");

        RuleFor(x => x.ImageUrl).NotEmpty().WithMessage("Resim Url Değeri Boş Bırakılamaz!");
    }
}
```

- ⇒ Mesela şifrelerle alakalı validasyon işlemleri ve hata mesajları, Create ve update sayfaları için gerekli validasyon işlemleri gibi tüm validasyon kontrolleri **Business katmanındaki Validators** klasöründe sınıflar oluşturularak yapılır ilk önce. Daha sonra controller tarafında bu validasyon sınıfları kullanılacak ama önce Businesssta oluşturuyoruz.

**Not:** Yalnız Controller tarafına geçmeden bu validasyon sınıfını **WebUI katmanına** taşımamız gerekmektedir. Dolayısıyla **Program.cs** dosyasında bunu şu şekilde eklememiz gerekir Yalnız burada da hata vermemesi için :

- ⇒ **WebUI** Katmanına geldik. Burada **Dependencies** üzerine sağ tık yapıp **Manage NuGet Packages**'i açıyoruz ve **Browse** kısmına "**fluentvalidation**" yazıyoruz.
- ⇒ **FluentValidation.DependencyInjectionExtensions** paketini yükleriz.
- ⇒ Yükleme işlemi tamamlandıktan sonra **WebUI** katmanındaki **Dependencies** içerisine girip **Packages** bölümünden yüklenip yüklenmediğini kontrol edebiliriz.

Ardından **Program.cs** 'e şu şekilde ekleme yapıyoruz 😊

⇒ `builder.Services.AddValidatorsFromAssemblyContaining<SingerValidator>();`

⇒ Daha sonra ise controller kısmında valiasyonu kullanmak istediğimiz metodun içerisinde oluşturmuş olduğumuz Validasyon sınıfından bir nesne türeterek (new' leme) `Validate(parameter)` metodu içerisine kullanarak işlemlerimizi gerçekleştiririz. Detayı OneMusic Projesi İçerisindedir

```
public IActionResult CreateSinger(Singer singer)
{
    var validator = new SingerValidator();
    ModelState.Clear();
    var result = validator.Validate(singer);
    ...
}
```

## Identity : Kimlik işlemleri (login/register)

### Kütüphaneleri dahil etme

- ⇒ İlk olarak kütüphanelerimizi kurarak başlayalım.
- ⇒ İlk olarak **WebUI** katmanında **Dependencies** bölümüne sağ tık yapıp **Manage NuGet Packages** 'i açıyoruz. **Browse** kısmına **Identity Core** yazarız.
- ⇒ Ardından çıkan **Microsoft.AspNetCore.Identity** paketini yükleriz.
- ⇒ Ardından **EntityLayer** katmanını açıp **Dependencies** bölümüne sağ tık yapıp **Manage NuGet Packages** 'i açıyoruz. **Browse** kısmına **Identity Core** yazarız.
- ⇒ Ardından çıkan **Microsoft.AspNetCore.Identity.EntityFrameworkCore** ve **Microsoft.AspNetCore.Identity** paketlerini yükleriz.
- ⇒ Unutmayalım ki tüm bu paket yükleme işlemlerinde projemizin sürümü ile uygun sürümü yüklemeliyiz.

### Class'ları oluşturma ve Context'e , Program.cs'e dahil etme

Paketlerimizi yükledikten sonra EntityLayer Katmanına geliyoruz ve Entities klasörüne yeni bir class(entity) oluşturacağız. **AppUser** adında.



- ⇒ Aslında Identity kütüphanesi bizlere User sınıfları oluşturcak. Fakat otomatik oluşturulacak olan User class'ının içerisinde UserName, Email, Password gibi değerler olsa da mesela Ad Soyad gibi değerler yok.
- ⇒ Dolayısıyla biz fazladan property'ler eklemek istediğimiz zaman (register'da sıklıkla bulunurlar mesela ad-soyad) **custom bir AppUser class'ı** oluşturuyoruz.

```
public class AppUser : IdentityUser<int> // int: primary key değeri
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public string? ImageUrl { get; set; }
}
```

- ⇒ **AppUser** Sınıfı **IdentityUser** 'dan miras alacaktır.
- ⇒ **ImageUrl** property'si zorunlu olmasın istenirse boş geçilebilir diye **string** ifadesinin sonuna **?** gelir.
- ⇒ Ardından **AppRole** sınıfı da ekleriz. Ve **AppRole** Sınıfı **IdentityRole** 'dan miras alacaktır. AppRole class'ının içerisine property yazmayız. Bu sınıfının oluşturulma sebebi zaten Primary Key değerinin değiştirilebilir olması sebebiyledir.
- ⇒ Identity'i projemize dahil ettik dolayısıyla DataAccesLayer Katmanındaki → **Context** sınıfında da **OneMusicContext** artık DbContext'ten değil IdentityDbContext'ten miras **almalıdır**. Bunu da değiştiriyoruz:

```
public class OneMusicContext : IdentityDbContext
```

IdentityDbContext'ten miras alacak. O sırada Microsoft.AspNetCore.Identity.EntityFrameworkCore'u dahil etti.

Ardından burada bir konfigürasyon yapmamız gereklidir, **IdentityDbContext** 'e AppUser, AppRole ve primary key değeri olan int'i dahil ederiz:

```
OneMusicContext : IdentityDbContext<AppUser,AppRole,int>
```

- ⇒ Ve tabiki **Program.cs**'e de Identity'i dahil etmemiz gereklidir.

```
builder.Services.AddIdentity<AppUser,AppRole>().AddEntityFrameworkStores<OneMusicContext>();
```

Ardından ise Entity ve DataAccess katmanlarında değişiklik yaptığımız için bunu SQL tarafına da yansıtmamız gerek ve dolayısıyla;

- ⇒ **Package Manager Console** 'u açıyoruz ve **add-migration mig\_identity\_added** diyor ve ardından **update-database** komutlarımızı yazarak migrationumuzu eklemiş ve database 'i update etmiş oluyoruz. Zaten sql tarafında tablolarımıza baktığımızda AppUser ve AppRole ile başlayan bir çok tablo oluşturduğunu görcez.

## Register ve Login işlemleri - Controller tarafında

- ⇒ Controller tarafın da asenkron yapılar kullanmaya başlayacağız bu aşamadan itibaren.
- ⇒ Şimdi normalde biz Controller tarafında çağırabilmek ve constructor'unu verebilmek için DataAccess ve Business Katmanında classlar ve service'ler yazıyorduk. Daha sonra onları çağırıp parametrelerini field'larına atıyor ve field'ları metotlar içerisinde kullanıyorduk (Örneğin; \_aboutManager gibi.. yukarılarda detaylarından bahsedilmiştir.)
- ⇒ Fakat Identity işlemleri için bunu yapmamıza gerek kalmayacak çünkü **Identity** kütüphanesinin bizlere sunduğu **\_userManager** field'ini kullanabileceğimiz **userManager** adında bir service'i var zaten. Aşağıdaki şekilde Controllerlarda

```
public readonly userManager<AppUser> _userManager;
```

```
public RegisterController(userManager<AppUser> userManager)
{
    _userManager = userManager;    ➔ Constructor
}
```

Mesela Diğer Controller yapılarında metotlar şu şekildeyken;

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

HttpPost kısmında Identityde **async** (asenكرون ) ve **Task<>** kullanılır;

```
[HttpPost]
```

```
public async Task<IActionResult> Signup()
{
    return View();
}
```

⇒ Controller’da çağırabilmemiz için **WebUI** Katmanında Custom bir Model oluştururuz:

#### ⇒ RegisterViewModel

⇒ Hatırlarsak AppUser sınıfı oluşturmuştuk ki eklemek isteyeceğimiz fazladan propertyleri geçebilelim. Bunun için bir Model Class’ı da oluşturmamız gerekiyor ve propertylerimizi tanımlamamız gerekiyor daha sonra controllerda çağırmak üzere;

```
public class RegisterViewModel
{
    public string Name { get; set; }
    public string SurName { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }

    [Compare("Password", ErrorMessage = "Şifreler birbiri ile uyumlu değil")]
    public string ConfirmPassword { get; set; }
} → Compare metodu kıyaslama metodudur. Password ile ConfirmPassword ‘ü kıyaslayacak.
```

⇒ Daha sonra ise RegisterControllerda aşağıdaki şekilde post işlemleri için bu modeli çağırıp bundan parametre türetebiliyoruz:

#### [HttpPost]

```
public async Task<IActionResult> Signup(RegisterViewModel model)
{
    AppUser user = new AppUser // AppUser sınıfını newledik
    {
        Email = model.Email, //soldaki Identityden, sağ RegisterViewModelden
        UserName = model.UserName,
        Name = model.Name, //soldaki AppUserdan, sağ RegisterViewModelden
        Surname = model.SurName
    };

    if(model.Password == model.ConfirmPassword) → eşitleme koşulu
    {
        var result = await _userManager.CreateAsync(user, model.Password);
    }
}
```

//AppUser türünde bir parametre bekliyor çünkü yukarıda UserManager içerisinde AppUser yazmıştık. Burada da Appuser türünde user parametresini geçirdik. O nedenle direk modeli vermedik parametre olarak, AppUserdaki parametreyi Model'deki Password e atadık. Ayrıca **await keyword'ü** de önemlidir.

```
        if (result.Succeeded) // sonuç başarılı olursa
        {
            return RedirectToAction("Index", "Login"); //Logine at
        }
        foreach (var item in result.Errors) //Başarılı olmazsa hata mesajı
        {
            ModelState.AddModelError("", item.Description);
        }
    }

    return View();
}
```

Kullanıcı adı: Erhan01, Erhan0101. (Şifresi)

- ⇒ Hatırlarsak **Register** işlemi için controllerlarda **Identity** kütüphanesinin bizlere sunduğu **\_userManager** field'ini kullanmıştık, **UserManager** adında bir service'i vardı.
- ⇒ Aşağıdaki şekilde **Login** işlemi için de **LoginController**larda **SignInManager** service'ini ve ondan türetilen **\_signInManager** field'i'ni kullanıyoruz.

```
private readonly SignInManager<AppUser> _signInManager;

public LoginController(SignInManager<AppUser> signInManager) //controller
{
    this._signInManager = signInManager;
}
```

- ⇒ Burada da aynı registerda yaptığımız işlemleri yapıyoruz aslında bir **LoginViewModel** oluşturduk, ardından **asenkron** olarak modeli çağırdık ve metodun içine parametre olarak verdik. Yani özetle registerda yaptığımız işlemleri yaptık burada da tek fark Registerda AppUser sınıfını newlemiştik, LoginControllerlarda newlememize gerek kalmadı, detayları OneMusic Projesi LoginControllerlarda...

## Proje Seviyesinde Authorize İşlemi

Normalde MVC’de Authorize işlemini ASP.NET MVC Projelerimizde Authorize işlemini gerçekleştirmek için her Controller’da tek tek yapardık bazen de proje seviyesinde yapardık. Şimdi burada hem MVC5 için hem de .Net Core için nasıl yapıldığını anlatacağım.

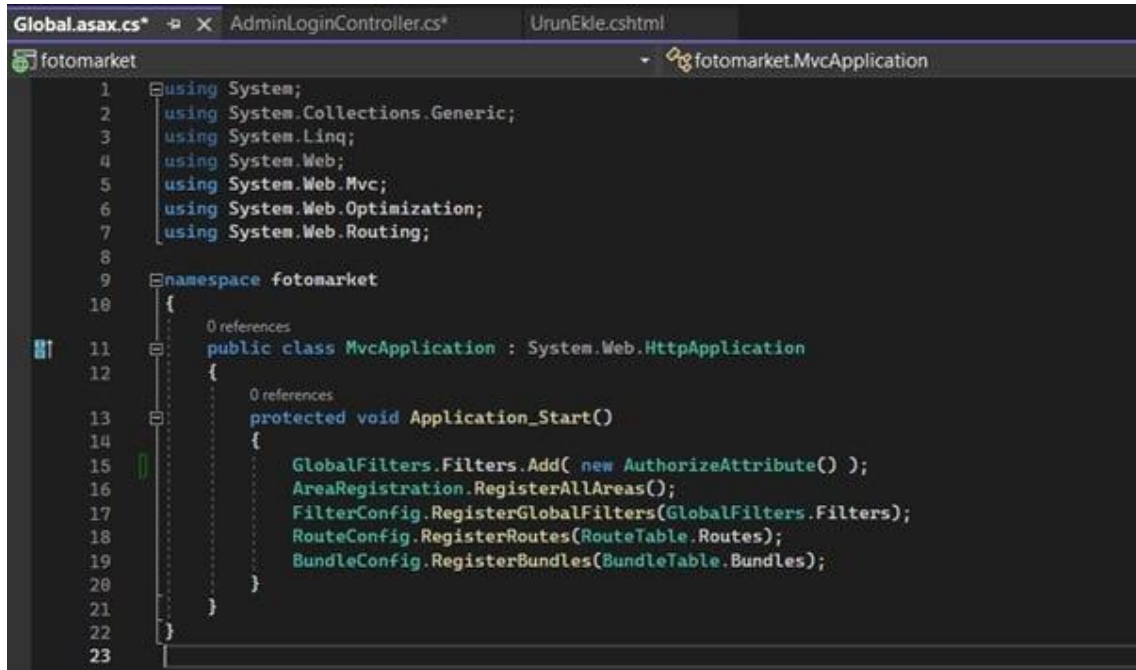
### MVC5 ‘te Proje Seviyesinde Authorize İşlemi:

⇒ Öncelikle Web.Config dosyamızı açalım.

```
<system.web>
  <compilation debug="true" targetFramework="4.8.1" />
  <httpRuntime targetFramework="4.8.1" />
  <authentication mode="Forms">
    <forms loginUrl="/AdminLogin/Index/"></forms>
  </authentication>
</system.web>
```

- ⇒ Ardından <system.web> etikenin içerisine yukarıdaki gibi kodları ekleyelim.
- ⇒ **loginUrl** kısmına sisteme **authentication(kimlik doğrulama)** yapacağımız formun adresini girelim.
- ⇒ Bu işlemin ardından projemizde bulunan **Global.asax** adlı dosyayı açalım.
- ⇒ Ardından **Application\_Start** fonksiyonuna şu kod satırını ekleyelim.

***GlobalFilters.Filters.Add( new AuthorizeAttribute() );***



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web;
5 using System.Web.Mvc;
6 using System.Web.Optimization;
7 using System.Web.Routing;
8
9 namespace fotomarket
10 {
11     public class MvcApplication : System.Web.HttpApplication
12     {
13         protected void Application_Start()
14         {
15             GlobalFilters.Filters.Add( new AuthorizeAttribute() );
16             AreaRegistration.RegisterAllAreas();
17             FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
18             RouteConfig.RegisterRoutes(RouteTable.Routes);
19             BundleConfig.RegisterBundles(BundleTable.Bundles);
20         }
21     }
22 }
23
```

⇒ Böylece tüm Controllerlarımız için tek tek **[Authorize]** yazmaktan kurtulmuş olduk. Fakat Admin Login Sayfası veya dışardan gelen her kullanıcıya göstermek istediğim Home Sayfasında bu işlemi gerçekleştirmek istemiyorum. Peki bunun için ne yapmalıyım? Bunun için ilgili controllerların başına **[AllowAnonymous]** komutunu ekleyeceğim.

```
1 using fotomarket.Models;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Web;
6 using System.Web.Mvc;
7 using System.Web.Security;
8
9 namespace fotomarket.Controllers
10 {
11     [AllowAnonymous]
12     public class AdminLoginController : Controller
13     {
14         // GET: AdminLogin
15         fotomarketEntities data = new fotomarketEntities();
16
17         [HttpGet]
18         public ActionResult Index()
19         {
20             return View();
21         }
22     }
23 }
```

⇒ Böylece ilgili sayfalar **authorize** işlemine tabi tutulmamış olacaktır.



⇒ Gördüğümüz gibi sadece adminin erişebileceği AynasizMakineler/Index sayfasına artık dışardan erişilememektedir.

## .Net Core'da Proje Seviyesinde Authorize İşlemi:

⇒ Burada direk **Program.cs** dosyasına gidiyoruz.

```
builder.Services.AddControllersWithViews(option =>
{
    option.Filters.Add(new AuthorizeFilter());
});
```

⇒ Bu şekilde proje seviyesinde bütün Controllerlarımıza **Authorize** eklemiş olduk.

⇒ Fakat sadece burasıyla bırakırsa **Loginle kayıt olan tüm sayfalara erişim kesilmiş oluyor**. Bu sorunun giderilmesi için **Login** ve **Register** controllerlarına **AllowAnonymous** ekleriz ki onlara erişim sağlayabilelim.

*[AllowAnonymous] //authorize'dan muaf*

```
public class RegisterController : Controller
{
    ...
}
```

*[AllowAnonymous] //authorize'dan muaf*

```
public class LoginController : Controller
{
    ...
}
```

⇒ Şimdi bu durumda proje de Login ve Register indexleri dışında diğer herhangi bir index'ten projeyi çalıştırmaya kalkarsak Unauthorized hatası alırız. Login ile giriş yapmadan sayfalara erişemeyiz.

- ⇒ Fakat yapmamız gereken bir şey daha kaldı. Şimdi bu işlemlerden sonra biz diyelimki AdminAbout Index'inden projeyi çalıştırmayı deneyince olması gerektiği gibi unauthorize hatası veriyor ama yönlendirdiği url de hata var yani Account/Login.. 'li bir url'e atıyor. Fakat böyle olmamasını istiyoruz. Bizi **/Login/Index** Url'ine atması gerek. Bunun için **Program.cs**'e eklenmesi gereken bir **konfigürasyon** daha var.

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Login/Index";

    // bu işlemi mvc de web.config de yapardık, net core da ise program.cs de yaparız
});
```

- ⇒ Bu işlemi de yaparak **LoginPath** yani giriş yaparken olması gereken Url'i de belirlemiş olduk. Aynı bu şekilde Çıkış Yapma (**LogoutPath**), **SessionStore**, UnAuthorize hata mesajlarının(böyle bir sayfa bulunamadı) vs. konfigürasyonlarını/ayarlamalarını da burada yaparız.

## View Component yapısı

MVC de partiallar kullanırdık hatırlarsak, Asp.Net Core'da da Partial yapısı var ve kullanılıyor tabiki.

Ama Asp.Net Core'da bize sunulan bir View Component yapısı vardır. Çok daha isteklerimize uygun yapıları tasarlayabileceğimiz şekildedir.

Ayrıca Partial Viewde biz projeyi çalıştırabilirken, View Component'ta çalıştıramıyoruz.

- ⇒ WebUI katmanına sağ tıklayıp yeni bir klasör oluşturuyoruz.
- ⇒ Klasörün adına ViewComponents deriz ve bunun içerisine oluşturacağımız Viewlerin servisleri (class türünde) yer alır
- ⇒ ViewComponents içerisine AdminLayout klasörü oluşturup onun içerisine **\_AdminLayoutNavbar.cs** sınıfı oluştururuz.
- ⇒ **\_AdminLayoutNavbar : ViewComponent** 'ten miras alır



```

public class _AdminLayoutNavbar : ViewComponent
{
    private readonly UserManager<AppUser> _userManager;

    public _AdminLayoutNavbar(UserManager<AppUser> userManager)
    {
        _userManager = userManager;
    }

    public async Task<IViewComponentResult> InvokeAsync()
    {
        var user = await _userManager .FindByNameAsync(
            User.Identity.Name);
        ViewBag.username = user.Name + " " + user.Surname;
        return View();
    }
}

```

- ⇒ Yine **UserManager<AppUser> \_userManager;** şeklinde yapılır ve constructor oluşturulur. Diğer kısımlarla aynı burası.
- ⇒ **IViewComponentResult** türünde bir **Invoke()** metodumuz vardır. Tabi Asenkron yapıda olduğu için **public async Task<IViewComponentResult> InvokeAsync()** olarak kullanılır.

. . .

- ⇒ Daha Sonra **Shared** klasörüne gideriz ve **Components** adında klasör oluştururuz. Onun altına da **\_AdminLayoutNavbar** adında bir klasör daha oluştururuz.
- ⇒ Bu kısımda Html kodları yer alacağından dolayı Partialları aslında burada yapıyoruz. Yani Viewler oluşturuyoruz. Mesela bu işlem için:
- ⇒ **\_AdminLayoutNavbar** klasörüne sağ tıklayıp **add view** deriz (view oluşturcaz). **Razor View'i** seçeriz. Ardından çıkan ekranda partial olacağı için **Create a partial'i** seçeriz.
- ⇒ Fakat isim olarak oluşturacağımız tüm partiallara biz **default** adını veriyoruz. Dolayısıyla ismini de **Default** olarak değiştiririz.
- ⇒ Unutma Shared=> Components=> Klasör adı , ViewComponent içerisindeki Sınıf adıyla aynı olacak
- ⇒ Ardından Layout Index dosyasında(RenderBody'nin) bulunduğu kesip Partial olarak ayırdığımız kısma **@await Component.InvokeAsync("\_AdminLayoutNavbar")** ekleriz. (MVC5 te **@Html.Action("DefaultblablaPartial")** gibi çağırırdık, burda **@await** kullanıyoruz.)
- ⇒ Bu işlemi Admin paneline giriş yaptığımızda Navbar kısmında Giriş yapan kişinin ad soyad bilgilerinin gözükmesi için yaptık.