ASP.NET CORE-MVC

İçindekiler

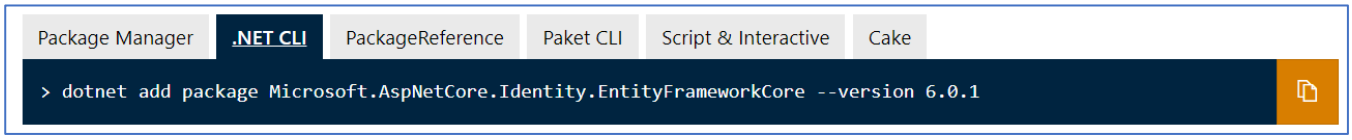
ÜYELİK İŞLEMLERİ

Üyelik Tablolarının Oluşturulması (Identity Kütüphanesinin Eklenmesi)

Üyelik işlemleri için Identity kütüphanesini kurmamız gerekecek. Bunun için google “Entity framework core Identity” yazalım.

https://www.nuget.org/packages/Microsoft.AspNetCore.Identity.EntityFrameworkCore

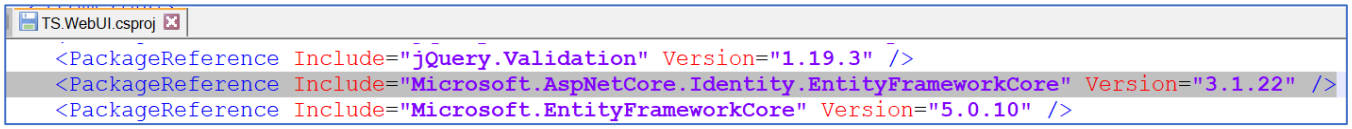
Kütüphanenin dotnet kurulum adresini alalım.



Bu son versiyon kütüphane çalışmadı. Bunun yerine daha eski olan şu versiyon yüklendi.Bu kütüphaneyi WebUI projesi içerisine kuracağız.

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 3.1.22

Bu versiyonun yüklendiğinin kontrolünü csproj uzantılı dosya içine bakarak yapalım.



Web projemizin içerisine "Identity" isminde bir klasör ekleyelim. Bunun içerisine sağ tuşa tıklayarak bir tane User class ı ekleyelim. Bu classı Identity.user sınıfından türeceğiz (`public class User:IdentityUser`). Bu sınıfın Namespace ini de yukarı ekleyelim (`using Microsoft.AspNetCore.Identity;`).

Veritabanında oluşacak olan user tabloları ve içerisindeki kolonlar IdentityUser dan gelecek. Bu sınıf üzerinde sağ tuşa tıklayıp "Go to definition" dersek sınıfın alt özelliklerini görmüş oluruz.

```
public class IdentityUser<TKey> where TKey : IEquatable<TKey>
{
    public IdentityUser();
    public IdentityUser(string userName);

    public virtual string SecurityStamp { get; set; }
    public virtual bool PhoneNumberConfirmed { get; set; }
    public virtual string PhoneNumber { get; set; }
    public virtual string PasswordHash { get; set; }
    public virtual string NormalizedUserName { get; set; }
    public virtual string NormalizedEmail { get; set; }
    public virtual DateTimeOffset? LockoutEnd { get; set; }
    public virtual bool LockoutEnabled { get; set; }
    public virtual TKey Id { get; set; }
    public virtual bool EmailConfirmed { get; set; }
    public virtual string Email { get; set; }
    public virtual string ConcurrencyStamp { get; set; }
    public virtual int AccessFailedCount { get; set; }
    public virtual bool TwoFactorEnabled { get; set; }
    public virtual string UserName { get; set; }

    public override string ToString();
```

Burada bir kullanıcı için gerekli temel alanların olduğunu görürüz. Şifre kısmı PasswordHash yani şifrelenmiş olarak oluşturulduğu dikkat çekmekte.

Fakat biz burada verilen bu alanlardan daha farklı alanları da eklemek isteyebiliriz. Dolayısı ile oluşturduğumuz sınıf içerisine bunları kendimiz ekleyelim. Dışarıdan gelen bu alanlarla birlikte hepsi çalışacaktır. Örneğin burada kullanıcının adı soyadı bilgisi yoktur. Bunları da kendi sınıfımıza ekleyelim. Sınıfımızın son hali şu şekilde olur.

WebUI>Identity>**User.cs**

```
using Microsoft.AspNetCore.Identity;

namespace TS.WebUI.Identity
{
    public class User:IdentityUser
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

    }
}
```

Identity klasörüne tek tuşa tıklayıp ApplicationContext isminde yeni bir sınıf ekleyelim. Bu sınıf veritabanı işlemlerimizi yapacak. Biz bunu daha önce yaptığımız gibi DbContext den türetmeyeceğiz. Identity kütüphanesi içinde özelleştirilmiş "IdentityDbContext" den türeteceğiz. Bu context bizim tanımladığımız User nesnesi ile çalışacak. Üstte namespace ni ekleyelim.

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

Daha sonra sınıf içine bir Constructor ekleyeceğiz. Dışarıdan bazı özellikler göndereceğiz. Bunlardan biri DbContextOptions<> göndereceğiz. Bunun parametresi ise ise ApplicationContext olacak. Parametre ismine ise Options diyeceğiz. Tanımladığımız options da base ni alacağız. Base() göndereceğiz. EntityFramework.core namespace sini de ekleyelim.

Identity>**ApplicationContext.cs**

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace TS.WebUI.Identity
{
    public class ApplicationContext:IdentityDbContext<User>
    {
        public ApplicationContext(DbContextOptions<ApplicationContext>options):base(options)
        {
        }
    }
}
```

Startup içinde bu tanımlamaları da yapmamız gerekiyor. Burada AppContext ismindeki metodumuza <ApplicationContext> vereceğiz. Daha sonra (options=>) göndereceğiz. Options.Sqlite() ile hangi provider kullanacağını belirtiyoruz. SqLite bizden bir parametre bekler. Bunu Data projesi içinde tanımladığımız connection stringi alacağız ('DataSource=TSDb'). Aynı veritabanını kullanabildiğimiz gibi farklı bir veritabanı da kullanabiliriz. Biz adresi aynı yazacağız ve böylece aynı veritabanı üzerine Identity tablolarını ekleyecek. İlgili eklenen satır aşağıdaki şekilde olacaktır.

```
    public class Startup
    {

    services.AddDbContext<ApplicationContext>(options=>options.UseSqlite("data source=TSDb"));
```

Ardından ikinci satırda kullanacağımız Identity metoduna User kullanıcı bilgisini yazıyoruz. Ardından ikinci parametre olarak da roll tabloları için kullanacağımız temel sınıf olan IdentityRole sınıfını yazıyoruz. TokenProvider ifadesinin geçtiği komutlar ile parola işlemlerini resetlerken benzersiz bir sayı üretmek için gerekli olacak. Parolasını resetlerken burada üretilen benzersiz sayı ile mail adresine göndereceğiz. O benzersiz sayı ile resetleme işlemini yapacak. Detayı ileride anlatılacak. Bahsedilen satır aşağıdaki şekilde olacaktır.

```
services.AddIdentity<User,
IdentityRole>().AddEntityFrameworkStores<ApplicationContext>().AddDefaultTokenProviders();
```

Yapmamız gereken işlemler bu kadar. Burada AddDbContext kullanacağımız söylüyoruz. Kullanacağı veritabanı ise Sqlite aracılığı ile belirtiyoruz. Ardından oluşturduğumuz AddIdentity sınıfını tanıtıyoruz. Biz bunlar daha önce oluşturduk fakat proje bunu tanımaz. O nedenle giriş aşamasında bunları tanıtmamız gerekir.

Burada giriş biz kullanacağımız user ve roll bilgilerini veriyoruz. Bu işlem için kullanacağımız Context belirtiyoruz ve ardından parola resetleme için gerekli benzersiz sayı üretecek olan token yapısını ekliyoruz.

Startup içinde Configure servis içine servislerimizi ekliyoruz fakat bunları kullan dediğimiz yer ise app.UseAutontication(); metodunu eklediğimiz yerdir.

Artık user sınıfımız var. Bu context yapısını kullanan tabloları veritabanına aktarmamız gerekiyor. Bunun için WebUI projesi içnde komut satırından daha önce yaptığımız gibi bir migration oluşturacağız.

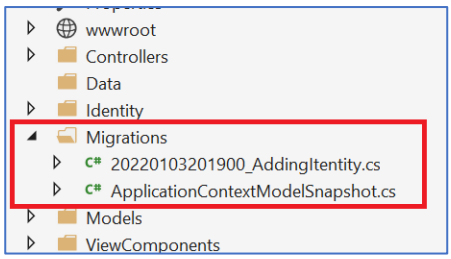
dotnet ef migrations add AddingItentity

webui> dotnet ef migrations add AddingItentity --context ApplicationContext (Yada bu şekilde kullan, denendi.)

bu şekilde yazdığımızda ilgili context leri tarayacak ve bunların içindeki yapılara göre bir migrations lar oluşturacak. Bu migrations lar içine bakacak olursak user tabloları, roll tabloları gibi diğer tablolarda olacaktır. Bu tabloları oluşturmak için gerekli scriptleri dosya içinde bulabiliriz. Komutu çalıştırmadan önce yazdığımız kodların VS içinde build etmeliyiz.

```
C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI>dotnet ef migrations add AddingItentity
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Artık migrations larımızın oluştuğunu Web projemiz içinde görebiliriz.



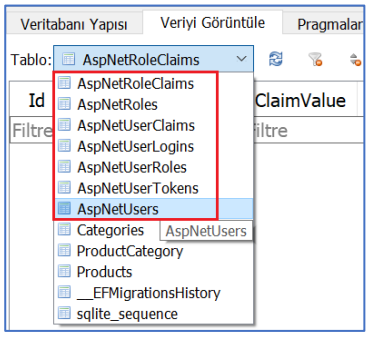
Artık bekleyen migrations larımızı içindeki tabloları database de oluşturmak için aşağıdaki komutu yazabiliriz.

dotnet ef database update

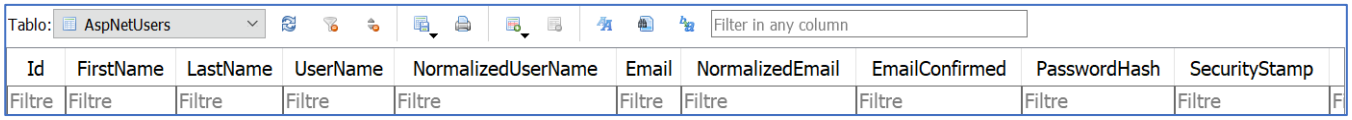
webui> dotnet ef database update --context ApplicationContext (bu şekilde kullan)

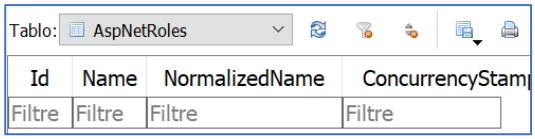
```
C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI>dotnet ef database update
Build started...
Build succeeded.
Done.
```

Veritabanımızı açtığımız zaman kullanıcı ile ilgili tablolarımızı görüyoruz.



Kullanıcı tablolarının ve Roll tablolarının içini inceleyelim.





Böylece bu uygulamada Identity kütüphanesine ait yapıları projemize ekledik. Şu anda gerekli alt yapımız hazır. Bu dersimizde ise bu yapıyı kendi projemize uygun hale getirmek için ayarlarını yapalım.

Identity Ayarlarının Yapılması

Kullanıcı sayfalarını oluşturmadan önce Identity ayarlarını yapalım. Bu ayarları yapmasakta olur. Varsayılan değerler ile kullanılabilir. İşi biraz daha iyi anlamak için yapalım. Bu bölümde yazacağımız kodlar AddItentiy() metodundan sonra gelmelidir. Kodların sıralamasında buna dikkat edilmeli.

Startup.cs içinde yukarıda yazdığımız kodların devamına yazalım. Services altında Configure metoduna IdentityOptions adında bir parametre gönderiyoruz. Ardından options üzerinden özelliklerimizi verelim. Bu özellikler belli gruplarda toplanmıştır. Örneğin options altında Password işlemleri bir gruptur. Burada kullanılan RequiredDigit seçilirse kullanıcının parolası içinde mutlaka bir sayısal ifade olmalıdır. True yerine false dersek sayısal bir ifade olması zorunlu olmaz. RequreLowerCase özelliği true yapılırsa parola içinde mutlaka küçük harf olmak zorunda olur. Yada UpperCase ayarladığımızda mutlaka büyük harf bulunması gerekir. RequireLenght=6 ayarlanınca en az 6 karakter olmalıdır. RequiredNonAlphanumeric =true yapılınca içinde sayı ve harfler dışında özel işaretlerin olması zorunlu olacaktır (@, %, -, _, & vs).

Diğer bir grup hesabın kilitlenmesi ile alakalı gruptur. `Lockout.MaxFailedAccessAttempts = 5;` dersek kullanıcı yanlış şifreyi maksimum 5 kez girebilir. `DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);` dersek hesap kilitlendikten sonra 10 dk zaman geçmesi gerekir. Bu özelliğin aktif olması içinde `AllowedForNewUsers = true;` özelliğini açmak gerekir .

Daha sonra user grubundan bahsedelim. `options.User.AllowedUserNameCharacters = "";` şeklinde yazdığımızda kullanıcı adı içinde olmasını istediğimiz karakterleri buraya yazabiliriz. Biz bunu kullanmayalım, yorum satırı yapalım. `RequireUniqueEmail = true;` dersek her kullanıcının birbirinden farklı mail adresi olmasını zorunlu kılarız. Benzer şekilde `RequireConfirmedPhoneNumber = true;` alanı da telefon onayı için kullanılır. Şimdilik bu iki özelliği false olarak kullanalım.

Son olarak Cookie (Çerez) ile ilgili ayarları yapalım. Bu kodlar yine AddItentity() metodunun kullanıldığı satırlardan sonra gelmelidir. Bu işlem için `ConfigureApplicationCookie()` metodunu kullanacağız. Bu metod yine dışarıdan bir option alacak. Cookie tarayıcı tarafından kullanıcının bilgisayarına bırakılan bazı bilgi dosyalarıdır. Tarayıcı daha sonra tekrar geldiğinde bu bilgileri kullanır. Örneğin biz bir arama yaparız. Aradığımız kelimelere uygun olarak

daha sonraları reklamları göstermek istediğimizde bu cookie bilgilerini kullanabiliriz. Yada kişinin daha önce siteye giriş yaptı bilgisayarına atılan cookie sayesinde bütün sayfalarda gezmesine müsaade edilir. Cookie lerin detaylarına ileride bakacağız.

Burada kullandığımız kodlardan LoginPath giriş yaptığımız sayfanın yolunu tanımlar, LogoutPath ise çıkış yaptığımızda gelecek sayfanın adresini belirler. AccessDeniedPath ise yetkimiz olmayan bir sayfa gittiğimizde çıkacak olan uyarı sayfasının adresini belirler. `SlidingExpiration = true;` komutu kullanılırsa bizim tarayıcımıza bırakılan cookie nin süresi 20 dk olur. Eğer 20 dk hiçbir işlem yapmazsak tekrar giriş yaptığımızda bizi login sayfasına gönderir. Her istek yaptımızda (servera gittiğimizde) süre tekrar 20 dk olarak sıfırlanır. `ExpireTimeSpan = TimeSpan.FromDays(3);` komutu ile varsayılan 20 dk süreyi 3 gün olarak ayarlayabiliriz.

Cookie ile ilgili bazı özellikleri verirken `HttpOnly = true,` ifadesini kullanırsak bizim cookie mize sadece tarayıcı sayfaları ulaşabilecektir. Bir javascript uygulaması ulaşamayacaktır. Biz bir sayfa için talepde bulunduğumuz istek servara gider. Cookie içindeki bilgilerde servera götürülür. Böylece server bu bilgileri tanıyıp tanımadığına bakarak devamındaki işlemlere karar verir. `Name = ".TS.Security.Cookie",` ifadesiyle cookie varsayılan yerine biz bir isim vermiş oluyoruz.

Identity ayarları için Startup içine eklediğimiz bu kodların tamamı aşağıda verilmiştir.

```
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationContext>(options=>options.UseSqlite("data
source=TSDb"));
            services.AddIdentity<User,
IdentityRole>().AddEntityFrameworkStores<ApplicationContext>().AddDefaultTokenProviders();

            //Kullanıcı ve Şifre ayarları
            services.Configure<IdentityOptions>(options =>
            {
                //Password
                options.Password.RequireDigit = true;
                options.Password.RequireLowercase = true;
                options.Password.RequireUppercase = true;
                options.Password.RequiredLength = 6;
                options.Password.RequireNonAlphanumeric = true;

                //Lockout
                options.Lockout.MaxFailedAccessAttempts = 5;
                options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
                options.Lockout.AllowedForNewUsers = true;

                //User
                //options.User.AllowedUserNameCharacters = "";
                options.User.RequireUniqueEmail = true;

                //SingIn
                options.SignIn.RequireConfirmedEmail = false;
                options.SignIn.RequireConfirmedPhoneNumber = false;

            });

            //Cookie ayarları
            services.ConfigureApplicationCookie(option => {
                option.LoginPath="/account/login";
                option.LogoutPath = "/account/logout";
                option.AccessDeniedPath = "/account/accessdenied";
                option.SlidingExpiration = true;
                option.ExpireTimeSpan = TimeSpan.FromDays(3);

                option.Cookie = new CookieBuilder {
```

```
                HttpOnly = true,
                Name = ".TS.Security.Cookie"
            };
        });

        //Data projesi içerisindeki IProductRepository çağrıldığında onun dolu versiyonu
olan EfCoreProductRepository gönderecek.
        services.AddScoped<IProductRepository, EfCoreProductRepository>();
        services.AddScoped<ICategoryRepository, EfCoreCategoryRepository>();

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

        //Bu yazı MCV deseni kullanımı için Controller Kullanımını Aktif ediyor.
        services.AddControllersWithViews();
    }
```

Üye Kayıt Sayfasının Oluşturulması

Şu aşamada üye kayıt yaparken Email onayı olmadan kayıt işlemini görelim. Bu işlem için `options.SignIn.RequireConfirmedEmail = false;` ayarını yapmıştık.

Öncelikle AdminController sınıfına yetkilendirilmiş bir kullanıcının erişebilmesini sağlamalıyız. Yani isteyen her kişi, linki biliyorsa adminin yapmış olduğu işlemleri yapamasın. Bunun için AdminController sınıfının üzerine [Autorize] komutunu ekliyoruz ve yukarıyada namespasini dahil ediyoruz.

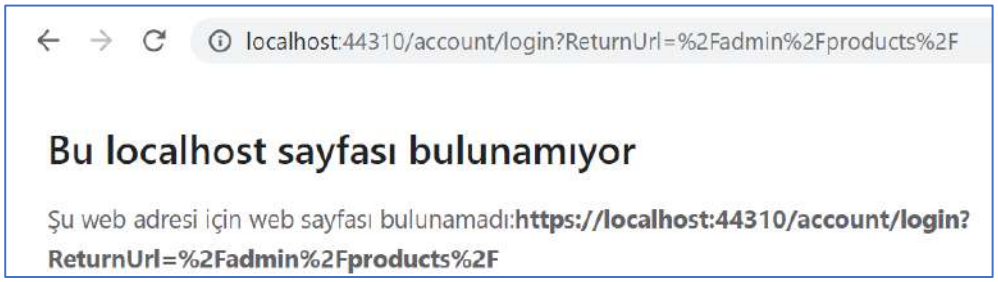
```
using Microsoft.AspNetCore.Authorization;
namespace TS.WebUI.Controllers
{
    [Authorize]
    public class AdminController : Controller
    {
```

Kodları test edelim. Siteyi çalıştırıp Admin ürünleri listeme yaptırdığımızda aşağıdaki hata sayfasını alırız.

Açıklamada UseAuthorization() ifadesini UseRouting() ile UseEndpoint() arasına eklememiz gerektiğini söylüyor. Burası Startup dosyasının içerisidir. Şimdi bu eklemeyi yapalım.

```
        app.UseAuthentication();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
```

Bunu ekledikten sonra tekrar denediğimizde bu sefer hata değişti. Bu durumda account/login sayfasına yani şifre giriş sayfasına gitmeye çalıştı ve eğer şifre girişini düzgünce yaparsak admin/products sayfasına göndereceğimizi söyledi.



Peki burada geçerli bir giriş yapmadığımızda /account/login sayfasına gitmemiz gerektiğini nerden bildi. Bu ayarı Startup.cs içinde aşağıdaki ayar ile yapmıştık. Bu ayarla kişiyi giriş yapma sayfasına göndermiş oluyoruz. Hemen altındaki accessdenied sayfası ise kullanıcı login girişi yapmış fakat yetkisi olmayan sayfaya gitmeye çalıştığında yönlendirilecek adresi tanımlamaktadır.

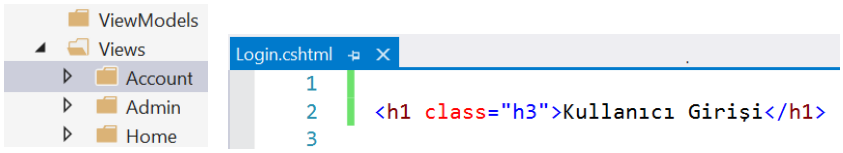
```
option.LoginPath="/account/login";
```

Bu durumda uygulamamıza bir account kontrolü eklememiz gerekiyor. Yani controllers klasörü içine AccountController.cs sınıfını oluşturalım. İçerisine de Login() metodumuzu ekleyelim.

WebUI>Controllers> **AccountController.cs**

```
using Microsoft.AspNetCore.Mvc;

namespace TS.WebUI.Controllers
{
    public class AccountController : Controller
    {
        public IActionResult Login()
        {
            return View();
        }
    }
}
```

Bu metodun gideceği View() sayfamızı oluşturalım. Şimdilik içerisinde basit bir başlık olsun.



Siteye girip admin i ilgilendiren bir sayfaya erişmeye çalıştığımızda yani AdminControler içinde bulunan metodlardan herhangi birine erişmeye çalışırsak (örn: Admin ürünleri listeleme sayfası gibi) bizi direk bu sayfaya yönlendirecektir .

Oluşan linki incelediğimizde iki kısımdan oluşuyor. Birincisi login sayfasının adresi (/account/login) ikinci bölüm eğer kullanıcı girişi yaparsak oradan yönlendirilecek adresin linki (ReturnUrl=/admin/products) kısmı

https://localhost:44310/account/login?ReturnUrl=%2Fadmin%2Fproducts%2F

AccountController da kullanıcıya ait bilgileri almak için Identity spacename içindeki UserManager sınıfı kullanılır. Bununla ilgili kodlarımız şu şekilde olur.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using TS.WebUI.Identity;

namespace TS.WebUI.Controllers
{
    public class AccountController : Controller
    {
        private UserManager<User> _userManager;
```

Buradaki _userManager nesnemizi kullanıcı ile ilgili işlemleri yaparken kullanacağız. Kullanıcımız ise web projemiz içinde oluşturduğumuz User sınıfıdır. Bu sınıfı Identity klasörünün içinde oluşturulmuştu.

Ayrıca Cookie olaylarını yönetmek için aşağıdaki kodları hemen altına ekleyelim. Bu koddan anlamamız gereken, controller içinde _signInManager şeklinde bir nesne kullanacağız. Bu nesne SingInManager sınıfından türetilmektedir. Bu sınıf ise dışırıdan <User> şeklinde bir nesneyi almaktadır. Dolayısı ile _signInManager nesneside bu User aşağılarda kullanacaktır.

```
private SignInManager<User> _signInManager;
```

Bu metodları Controller içine Injection işlemine tabi tutmamız gerekiyor. Yani dışarıdan gelen SingInManager sınıfı tipinde olan ve içerisinde User nesnesini kullanan signInManager şeklinde yazılan bu nesne controller içinde kullanılacak olan aynı tipte ve yine User tipini kullanan _signInManager nesnesine aktarmak için yapılan işlemdir.

```
public AccountController(UserManager<User> userManager, SignInManager<User> signInManager)
{
    _userManager = userManager;
    _signInManager = signInManager;
}
```

Artık Controllor sayfamızda Kullanıcı işlemlerini yapmak için UserManager sınıfımız ve Cookie işlemlerini yapmak için ise SignInManager sınıfımız vardır.

Bundan sonra kullanıcı kayıtı için Register() metodumuzu oluşturabiliriz. Bu metodun bir HttpGet versiyonu birde HttpPost versiyonu olacak. Post versiyonumuz sayfadan gelen RegisterModel i kullanacak. Bu modeli models klasörü içinde oluşturalım. İçerisinde aşağıdaki alanlar bulunsun. Bazı eklenen DataAnnotation (veri yazım usulleri) açıklamaları şu şekilde olacaktır. [Required] alanlarını doldurmak zorunludur. `[DataType(DataType.Password)]` yazım şekli password şeklinde gizli yazım usulü yapacaktır. `[Compare("Password")]` ifadesi üstteki password alanı ile aynı olup olmadığını kontrol edecek. `[DataType(DataType.EmailAddress)]` ifadesi mail adresi formatında yazımını kontrol edecek.

Models> **RegisterModel.cs**

```
using System.ComponentModel.DataAnnotations;

namespace TS.WebUI.Models
{
    public class RegisterModel
    {
        [Required(ErrorMessage ="Ad alanını doldurmak zorunludur. ")]
        public string FirstName { get; set; }

        [Required(ErrorMessage = "Soyad alanını doldurmak zorunludur. ")]
        public string LastName { get; set; }

        [Required(ErrorMessage = "Kullanıcı adını doldurmak zorunludur. ")]
        public string UserName { get; set; }

        [Required(ErrorMessage = "Şifre girilmesi zorunludur. ")]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Required(ErrorMessage = "Şifreyi tekrar yazmak zorunludur.")]
        [DataType(DataType.Password)]
        [Compare("Password",ErrorMessage ="Şifreler aynı değildir. ")]
        public string RePassword { get; set; }
```

```
        [Required(ErrorMessage = "Email girilmesi zorunludur. ")]
        [DataType(DataType.EmailAddress)]
        public string Email { get; set; }
    }
}
```

Şimdi Get tipindeki Register() metodumuzun gideceği View sayfasını oluşturalım. Views>Account>Register.cshtml dosyamızı ekleyelim. İçerisine bilgileri ProductCreat.cshtml den örnek olarak alıp düzenleyelim.

Views>Account>**Register.cshtml**

```
@model RegisterModel

<h1 class="h3">Üye Kayıt</h1>
<hr>
<div class="row">
    <div class="col-md-12">
        <div asp-validation-summary="All" class="row text-danger"></div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <form asp-controller="Account" asp-action="Register" method="POST">
            <div class="form-group row mb-3">
                <label asp-for="FirstName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="FirstName">
                    <span asp-validation-for="FirstName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="LastName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="LastName">
                    <span asp-validation-for="LastName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="UserName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="UserName">
                    <span asp-validation-for="UserName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="Password" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Password">
                    <span asp-validation-for="Password" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="RePassword" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="RePassword">
                    <span asp-validation-for="RePassword" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="Email" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Email">
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
            </div>
```

```
            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2">
                    <button type="submit" class="btn btn-primary">Kullanıcıyı Kaydet</button>
                </div>
            </div>
        </form>
    </div>
</div>
```

View() sayfamızda Post metodu ile gelen bilgiler kaydedilmek üzere Controller içinde gerekli olan kodları yazalım.

Aşağıdaki şekilde model yapısı içinde gelen bilgileri kaydetmek üzere sayfada üretilen user nesnesi üzerinden kaydediyoruz. Tabii öncesinde gelen model bilgisinin validation kontrolünü yapıyoruz. Burada tüm alanları aktarırken password alanı şifreleme şeklinde olduğu için onun aktarımını ayrı bir işlemle yapıyoruz. Bunun için asenkron bir metod kullanıyoruz. Bu kullanım için eklenen alanlar renkli olarak gösterilmiştir . CreateAsycr bizden bir user bir de parola bekliyor, bunları veriyoruz.

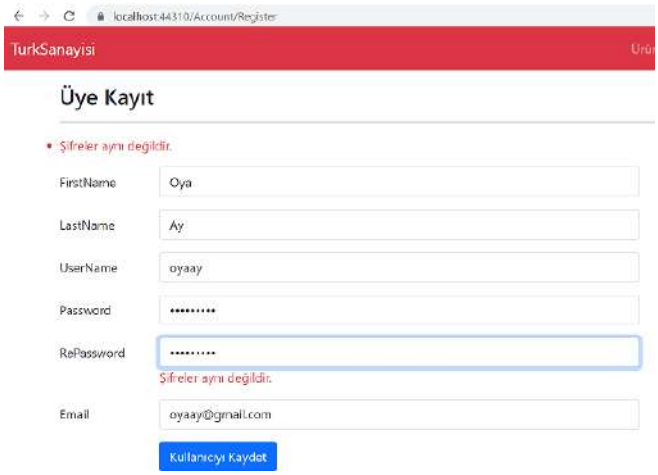
Elde edilen result sonucu başarılı ise kişiyi tekrar kendi şifresi ile giriş yapması için account/login sayfasına gönderiyoruz. Oraya gitmeden önce bir token oluşturuyoruz. Bu token ı kişinin mailine gönderip onaylatıyoruz. Bununla ilgili token oluşturma ayarlarını ve mail gönderme ayarlarını Startup.cs içinde yapmıştık. Fakat orada mail işlemini şimdilik kapattığımızdan, kişinin mail adresine token gönderme işini yapmayalım.

Controllers>**AccountController.cs**

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using TS.WebUI.Identity;
using TS.WebUI.Models;

namespace TS.WebUI.Controllers
{
    public class AccountController : Controller
    {
        private UserManager<User> _userManager;
        private SignInManager<User> _signInManager;

        public AccountController(UserManager<User> userManager, SignInManager<User> signInManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
        }
        public IActionResult Login()
        {
            return View();
        }

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

        [HttpPost]
        public async Task<IActionResult> Register(RegisterModel model)
        {
            if(!ModelState.IsValid)
            {
                return View(model);
            }

            var user = new User()
            {
                FirstName = model.FirstName,
                LastName = model.LastName,
                UserName = model.UserName,
                Email = model.Email
            };

            var result = await _userManager.CreateAsync(user,model.Password);
            if(result.Succeeded)
```

```
            {
                //token oluşturma işlemleri, şu aşamada iptal
                //maile token gönderme işlemleri, şu aşamada iptal
                return RedirectToAction("login","account");
            }
            ModelState.AddModelError("Password", "Şifre kurallara uygun değil. Harf ve sayılardan oluşacak. Min. 6 karakter olacak. Büyük küçük harfler bulunacak. İçerisinde özel karakter olacak.");

            return View(model);
        }

    }
}
```

Kodları çalıştırdığımızda şifreleri aynı girmezsek hata mesajı verdiğini görürüz.



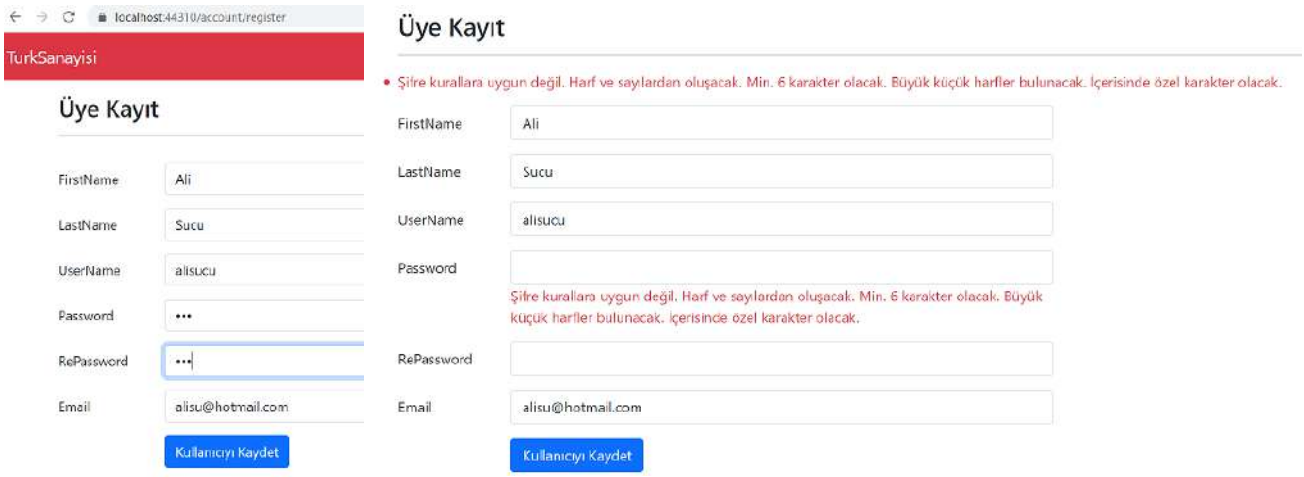
Burada şifreler aynı olmadığında hatayı gösterdi fakat şifrelerin Startup.cs içinde tanımlandığı şekilde olup olmadığının kontrolünü yapmadı. Bu konuda aşağıdaki gibi bir çok şifre için istediğimiz ayarlama vardı. Bunları kontrol etmedi.

```
//Password
options.Password.RequireDigit = true;
options.Password.RequireLowercase = true;
options.Password.RequireUppercase = true;
options.Password.RequiredLength = 6;
options.Password.RequireNonAlphanumeric = true;
```

Register() metodunun en sonuna aşağıdaki kodları yazarsak bu durumu kontrol etmiş oluruz. Bu ifadede "Password" alanında tırnakları boş bırakırsak formda en üstteki genel alanda gösterir. Aşağıda göstermez. Bu tip hataları istersek _layout.cshtml içinde kullandığımız mesaj sectionında gösterebiliriz.

```
    ModelState.AddModelError("Password", "Şifre kurallara uygun değil. Harf ve sayılardan oluşacak. Min. 6 karakter olacak. Büyük küçük harfler bulunacak. İçerisinde özel karakter olacak.");

    return View(model);
}
```

Kodları tekrar denersek şifreyi 123 olarak girdik.



Üye Giriş Sayfası

UserName ile Giriş Yapma

Bir önceki derslerimizde kullanıcı kayıt altyapısını hazırladık. Şimdi üye giriş sayfasına bakalım. Login işlemleri içinde bir tane Get özellikli ve bir tanede Post özellikli iki metod olacaktır. Bilgileri taşımak içinde LoginModel isminde bir model yapımız olsun.

Önce LoginModel yapımızı Models içinde oluşturalım. İçerisinde login sayfasında gireceğimiz bilgileri oluşturalım. Şifre girişi yaparken kullanıcı adı ve şifre alınması yeterli olacaktır. Kullanıcı adı olarak mail bilgisi kullanılacak ise Usarname yerine bu bilgi kullanılabilir. Girilecek bu iki alanı [Required] olarak tanımlayalım. Şifre girişinde karakterler gözükmemesi için DataType = "Password" olarak tanımlayalım.

Models> **LoginModel.cs**

```
using System.ComponentModel.DataAnnotations;

namespace TS.WebUI.Models
{
    public class LoginModel
    {
        [Required(ErrorMessage ="Kullanıcı adını giriniz")]
        public string UserName { get; set; }

        [Required(ErrorMessage = "Şifrenizi giriniz")]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

Login.cshtml sayfasını oluşturalım. İçeriğini örnek olarak Register sayfasından alalım.

Views>Account>**Login.cshtml**

```
@model LoginModel

<h1 class="h3">Üye Giriş</h1>
<hr>
<div class="row">
    <div class="col-md-12">
        <div asp-validation-summary="All" class="row text-danger"></div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <form asp-controller="Account" asp-action="Login" method="POST">
```

```
            <div class="form-group row mb-3">
                <label asp-for="UserName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="UserName">
                    <span asp-validation-for="UserName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="Password" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Password">
                    <span asp-validation-for="Password" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2">
                    <button type="submit" class="btn btn-primary">Siteye Gir</button>
                </div>
            </div>
        </form>
    </div>
</div>
```

Post tipindeki Login() metodunu oluşturalım. ModelState.IsValid değilse girilmeyen bir alan olabilir. Bu durumda sayfaya geri gönderelim ve geri giderken model içindeki bilgileri de tekrar geri götürelim.

Bir hata yoksa veritabanından user bilgisini alalım. Bakalım bu user var mı? Bu işi userManager nesnesi yapacak. Sorgulama yaparkan Id bilgisine göre, Username bilgisine göre yada mail adresine göre sorgulama yapabiliriz. Burada `FindByNameAsync` komutu ile kullanıcı adına göre sorgulama yapalım. Bu metod asenkron olduğu için başına await özelliğini ekleyeceğiz. Böylece kişiyi bulana kadar bekleyecek. Bunu yapınca üstte metod u da asenkron yapmamız gerekir ve metod adımız Task<> içine alacağız.

```
public async Task<IActionResult> Login(LoginModel model)

       var user = await _userManager.FindByNameAsync(model.UserName);
```

Eğer user null ise o zaman veritabanında bu kullanıcı adında birisi yoktur. Bu durumda tekrar sayfaya yönlendirip Message kutusu içinde bir mesaj verebiliriz. Bu işlem _layout sayfası içinden TempData içinden gönderiliyordu.

Yada sayfaya geri giderken ModelState içinde mesaj bilgisini verebiliriz. ModelState kullanırken herhangi bir model alanı adını yazmazsak (Username yada Password gibi) üstteki “All” özelliğinin olduğu yerde mesaj gösterilecektir.

```
var user = await _userManager.FindByNameAsync(model.UserName);

if(user==null)
{
       ModelState.AddModelError("", "Bu kullanıcı adı ile daha önce bir hesap
oluşturulmamıştır");
}
```

Eğer bu hatalar oluşmadıysa artık kullanıcıyı kabul edeceğiz ve sayfalarda gezmesine müsaade edeceğiz. Bu işlem için kişinin bilgisayarına `_signInManager.PasswordSignInAsync` aracılığı ile bir Cookie bırakacağız. Bu metodun **ilk iki** parametresi kullanıcıadı ve şifre olacaktır. **3.** Özellik IsPersistent bool türünde bir özelliktir. Tarayıcı kapandığında cookie nin silinip silinmeyeceğini gösterir. Bunu false yaparsak tarayıcı kapanınca cookie silinir. Startup içinde verilen süre kadar beklemez. **4.** Özellik hatalı giriş yapacağı sayıyı aştığında hesap kapansın mı

kapanmasın mı onun ayarıdır. False dersek hesap kapanmaz. Kaç defa denemeye müsaade edeceğimiz ise startup içinde ayarlamıştık. Kısaca her ikisini de true yaparsak startup içindeki ayarlara uyacaktır. False dersek bu ayarlara uymayacaktır. Şimdilik bunları false olarak kullanalım.

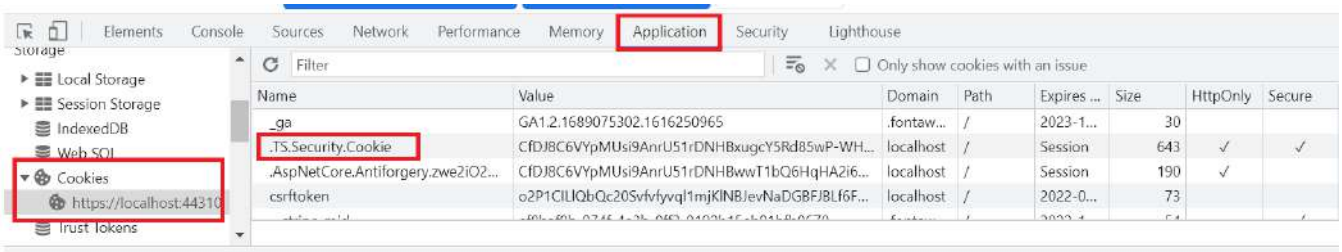
```
var result = await _signInManager.PasswordSignInAsync(model.UserName, model.Password, false,
false);
```

Sonuçta şifre girişi başarılı ise Home/Index sayfasına yönlendirebiliriz. Bu adresleri yazarken önce Metod (Index), sonra Action (Home) nın yazıldığına dikkat edin.

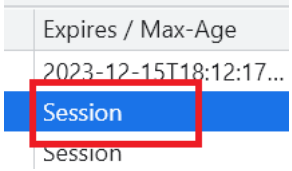
```
if(result.Succeeded)
{
    return RedirectToAction("Index", "Home" );
}
```

Kodları denediğimizde olmayan bir kullanıcı ile giriş yaparsak hata mesajını alıyoruz. Doğru şifre ile girersek ana sayfaya yönlendirmektedir. Bundan sonra artık Admin sayfalarına girmeye çalıştığımızda sorun çıkarmayacaktır.

Sayfada F12 ye tıkladığımızda Application altında Cookies altında daha önceden verdiğimiz Cookie adını görürüz (TS.Security.Cookie).



Cookie nin son zamanı burada tanımlanmadı. Çünkü biz Cookie atarken 3. Parametreyi False olarak tanımladık (`var result = await _signInManager.PasswordSignInAsync(model.UserName, model.Password, false, false);`). Bu durumda tarayıcı kapandığında Cookinin süresi bitmiş olacak.



F12 açtığımız sayfadan Cookie nin olduğu satırı silersek bizi tekrar Login sayfasına gönderecektir.

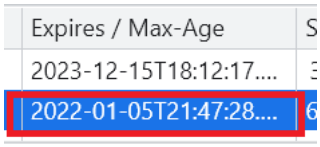
Şimdi Cookie attığımız 3. Parametreyi true yapalım.

```
var result = await _signInManager.PasswordSignInAsync(model.UserName, model.Password, true,
false);
```

Startup içinde cookie süresini 2 dk olarak ayarlamıştık.

```
option.ExpireTimeSpan = TimeSpan.FromMinutes(2);
```

Yeniden çalıştırıp F12 ile cookie yi kontrol edersek bu sefer süreyi görürüz.



Email Adresi ile Giriş Yapma

Şu ana kadar yaptığımız girişte UserName kullandık. Ama bir çok site mail adresi ile giriş yapılmasını ister. Bunun için LoginModel içinde yapacağımız değişikliklere bakalım. Eski ve yeni kodlar aşağıda verildiği şekilde olur.

```
    public class LoginModel
    {
        //[Required(ErrorMessage ="Kullanıcı adını giriniz")]
        //public string UserName { get; set; }
        [Required(ErrorMessage = "Mail adresinizi giriniz")]
        [DataType(DataType.EmailAddress,ErrorMessage ="Mail formatı yanlıştır")]
        public string Email { get; set; }
```

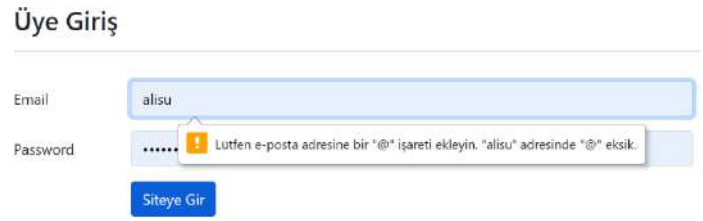
Kontroller içindeki bulunan Post tipindeki Login() metodunda değişecek satırlara bakalım.

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel model)
{
       //var user = await _userManager.FindByNameAsync(model.UserName);
       var user = await _userManager.FindByEmailAsync(model.Email);
//***

//var result = await _signInManager.PasswordSignInAsync(model.UserName, model.Password, true,
false);
var result = await _signInManager.PasswordSignInAsync(user, model.Password, true, false);
```

Login.cshtml içinde değişecek kısımlar ise aşağıdaki şekilde olur.

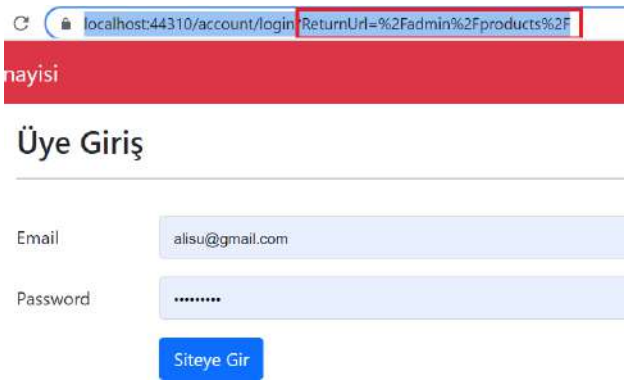
```
@*<div class="form-group row mb-3">
    <label asp-for="UserName" class="col-sm-2 col-form-label"></label>
    <div class="col-sm-10">
        <input class="form-control" asp-for="UserName">
        <span asp-validation-for="UserName" class="text-danger"></span>
    </div>
</div>*@
<div class="form-group row mb-3">
    <label asp-for="Email" class="col-sm-2 col-form-label"></label>
    <div class="col-sm-10">
        <input class="form-control" asp-for="Email">
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

Kodları deneyelim.



Login İşlemi Sonrası Yönlendirme (ReturnUrl işlemi)

Şu anda mail ile giriş yapabiliyoruz. Login girişi karşımıza ne zaman geldi. Biz admin/products ları görmek için linke tıkladığımızda yetkimiz olmadığı için Login sayfasını karşımıza getirdi. Buradan giriş yaptıktan sonra bizi "admin/products" lara götürmesi lazım fakat "home/index" sayfasına yönlendirmektedir. Bu durumu düzeltelim. Yani yetkimiz olmayan hangi alana girmeye çalışırsak bizi login den sonra oraya yönlendirsin.



Bu yönlendirmeyi yapmak için LoginModel içerisine aşağıdaki satırı ekleyelim

```
public class LoginModel
    {

     public string ReturnUrl { get; set; }
```

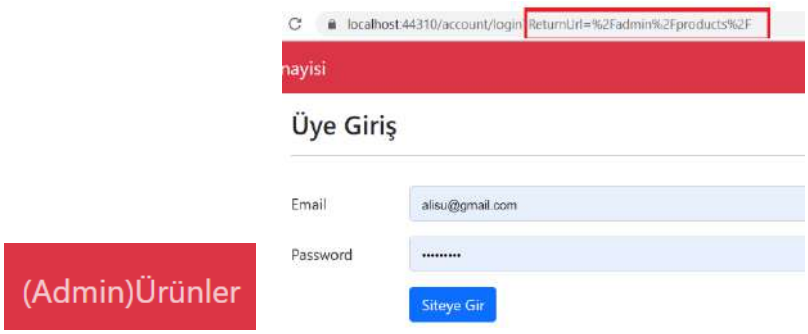
Kontroller içinden View Login sayfasına giderken Get metodlu Login() action ı kullanılıyordu. Dolayısı ile sayfaya giderken hangi Url adresi isteğiyle buraya geldiğimizi metod girişinde aşağıdaki şekilde alabiliriz. Fakat önceden View sayfasına giderken model yapımız içinde herhangi bir bilgi götürmüyorduk. Dolayısı buradan View sayfasına giderken model yapısı içinde ReturnUrl bilgisini de götürmeliyiz. Böylece View sayfasında gizli alan içerisinde Url bilgisini saklarız. Giriş işlemini yapıp Post Login() metoduna geldiğimizde de model içinde gizli alan olarak eklediğimiz bilgileri kullanarak başarılı girişten sonra gerekli yönlendirmeleri yaparız. (Not: Burada returnUrl=null şeklindeki kullanım şunu anlatıyor. Her zaman bir linkten buraya gelinmeyecektir. O yüzden eğer değişken boş gelirse metod girişinde de link boş olarak yüklenecektir)

```
    [HttpGet]
    public IActionResult Login(string returnUrl=null)
    {
        return View(new LoginModel
        {
            ReturnUrl = returnUrl
    });
    }
```

View sayfamızda bu Url bilgisini gizli alan içinde saklayalım.

```
<input type="hidden" name="ReturnUrl" value="@Model.ReturnUrl" />
```

Kodlarımızın buraya kadar olan kısmını deneyelim. (Admin)Ürünler linkine tıkladığımızda sayfa güvenli sayfa olduğundan bizi Login sayfasına yönlendirirken ReturnUrl linkini oluşturacaktır. Bunu adres satırında görebiliriz.



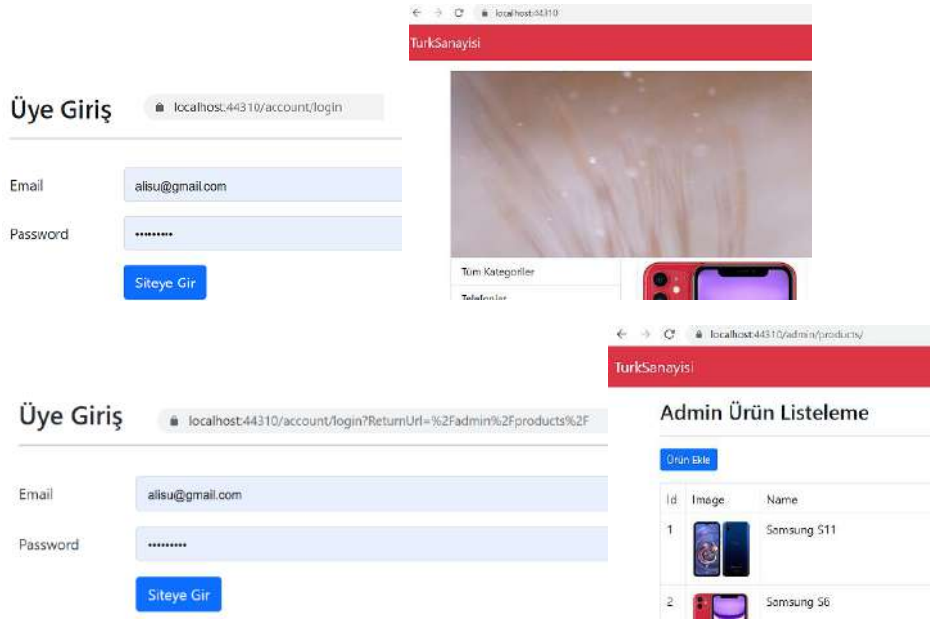
Kaynak kodlarına baktığımızda Login.cshtml sayfası içinde Url bilgisinin geldiğini görebiliriz.

```
<div class="row">
    <div class="col-md-8">
        <form method="POST" action="/Account/Login">
            <input type="hidden" name="ReturnUrl" value="/admin/products/" />
```

Artık giriş butonuna tıkladığımızda Post metoduna gideceğiz. Oraya model içinde hidden kısımdan bilgiler alınıp götürülecektir. Ve bizde giriş başarılı ise gerekli yönlendirmeyi url ye göre yaparız. Aşağıdaki kodlarda girişi başarılı ise model içinden gelen adrese yönlendirilecek. Fakat bu adres her zaman dolu gelmeyebilir. Dolayısı ile eğer null gelirse bu sefer sitenin ana sayfasına gitmesi için (ReturnUrl??"~/"); şeklinde bir yazım kullanıldı. İki tane soru işareti "Eğer null a eşitse" anlamındadır.

```
var result = await _signInManager.PasswordSignInAsync(user, model.Password, true, false);

if (result.Succeeded)
{
    return Redirect(model.ReturnUrl??"~/");
}
```

Kodları denersek login e direk giriş yaparsak ana sayfaya gidiyor. Admin sayfalarından yönlendirilsek, giriş sonrası admin sayfalarına gidiyor.



Kullanıcı Oturumunu Kapatma

Kullanıcı çıkışını yapmak için NavBar ı biraz düzenleyelim. NavBar içindeki Sepet gibi linkler giriş yapan kullanıcılara, Admin ile alakalı linkler ise admin girişi yaptığında gösterilmelidir. Ürünler ise direk kullanıcılara gösterilebilir. NavBar'ın sağ tarafında ise Kullanıcı Girişi (LogIn), yada Kulanıcı çıkış (LogOut) linkleri bulunmalıdır. Ayrıca kişinin kullanıcı adı da sağ tarafta gösterilmelidir.

Shared klasörünün altında _NavBar.cshtml dosyamızı açıp linklerimizi düzenleyelim. Navbar içinde iki tane listemiz olsun (`<ul class="navbar-nav ml-auto">`). Bir tanesi sola yaslı olsun yani "ml-auto" bu işlemi yapar. Diğeri ise sağa yaslı olsun. Class ismi "mr-auto" olsun. Sağa yaslı olan içinde kullanıcı giriş çıkış butonlarımız olsun. Son hali şu şekilde olabilir.

Sayfada ön bir deneme yaparsak linklerimiz aşağıdaki şekilde oldu. Sağ tarafta üç link aynı anda gözükmeyecek. Kullanıcı giriş yapmışsa "Çıkış yap" linki gözükmeli. Yada tam tersi olmalı. Giriş yapmışsa "Üye ol" linki de çıkmamalı.

Şimdi bu kontrolü yapmak için sayfa içine kodlama yapalım.

`@if(User.Identity.IsAuthenticated)` ifadesiyle kullanıcının giriş yapıp yapmadığını kontrol edebiliriz. Kullanıcı giriş yapmışsa, LogOut menüsünü kullanıcıya gösterelim. LogOut un yanına bir tane daha <li> etiketi ekleyip onun içinde de Kullanıcı adını (username) gösterelim. Bunun için `@User.Identity.Name` ifadesini kullanalım.

NavBar'ın sol tarafındaki linklerden ürünleri herkes görebilsin. Fakat yanındaki Sepet ve Admin işlemlerini ise sadece Login olanlar görebilsin. Tabi admin linklerinin rollere göre de düzenlenmesi gerekecek. Bunları da ileride yapalım.

_NavBar sayfamızın kodlarının son hali aşağıdaki şekilde olur.

Shared>**_navBar.cshtml**

```
@********NAVBAR*************@
<nav class="navbar bg-danger navbar-dark navbar-expand-sm">
    <div class="container-fluid">

        <ul class="navbar-nav ml-auto">
            <li class="nav-item">
                <a href="/" class="navbar-brand">TurkSanayisi</a>
            </li>

            <li class="nav-item">
                <a href="/products" class="nav-link">Ürünler</a>
            </li>

            @if (User.Identity.IsAuthenticated)
            {
                <li class="nav-item">
                    <a href="" class="nav-link">Sepet</a>
                </li>

                <li class="nav-item">
                    <a href="" class="nav-link">Sipariş</a>
                </li>

                <li class="nav-item">
                    <a href="/admin/products/" class="nav-link">(Admin)Ürünler</a>
                </li>

                <li class="nav-item">
                    <a href="/admin/categories/" class="nav-link">(Admin)Kategoriler</a>
                </li>
            }


        </ul>

        <ul class="navbar-nav mr-auto">
            @if (User.Identity.IsAuthenticated)
            {
                <li class="nav-item">
                    <a href="/account/manage" class="nav-link">
                        @User.Identity.Name
                    </a>
                </li>
                <li class="nav-item">
                    <a href="/account/logout" class="nav-link">Çıkış Yap</a>
                </li>
            }
            else
            {
                <li class="nav-item">
                    <a href="/account/login" class="nav-link">Giriş Yap</a>
                </li>
```

```
                <li class="nav-item">
                    <a href="/account/register" class="nav-link">Üye Ol</a>
                </li>
            }
        </ul>


    </div>
</nav>
```

Kodları denediğimizde kullanıcı girmeden önde sadece ürünler linkini gösterir. Giriş yaptığımızda ise diğer linklerde gelir.

Giriş yaptıktan sonra bilgisayarımıza Cookie atılacaktır. F12>Application>Cookies yolunu takip ederek atılan Cookie görebiliriz. Bu Cookie silersek LogOut olmuş oluruz. Bu işlemi ise LogOut butonundan (çıkış yap) butonundan yürüteceğiz.

Şimdi Logout işlemi için AccountController altında bir asenkron Logout metodu oluşturalım. Bu metod içinde basit bir cookie silme komutunu kullanıyoruz. Ardından ana sayfaya gidiyoruz. Ana sayfaya giderken iki şekilde gidebiliriz. Ya direk adres yazarız yada home controller içinde index metoduna yönleniriz. Metod adının (index) önce yazıldığına dikkat edin.

```
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return Redirect("~/");
    //return RedirectToAction("index","home");
}
```

Form Güvenliğini Sağlama (CSRF Token Kullanımı)

Diyelim bir kullanıcı banka hesabına giriş yapmak istiyor. Bu girişi yaptığında kullanıcının bilgisayarında Cookie oluşturulur. Banka serverlarında ise Session oluşturulur. Session ile cookie haberleştiği sürece giriş işlemi devam etmiş oluyor ve kullanıcı başka sayfalar üzerinde geçiş yapabiliyor.

Burada bir problem yoktur. Peki bir B kullanıcı A kullanıcının cookie sini çalmış olsa bu internet bankacılığına girebilir mi yada bu haberleşmeyi yapabilir mi? Bunu engellemek için ne yapılıyor, şimdi ona bakalım.

Bu iş önce Startup dosyasının içerisinde Cookie yapılandırması yaptığımız bir kod kısmı vardı. Bu kısmın içerisine aşağıdaki özelliği ekleyebiliriz. Eklediğimiz bu özellik, cookie ile session haberleşirken sadece cookie içindeki bilgiye bakmaz aynı zamanda adres bilgisini de kontrol eder. Böylece B kişisi ilgili cookiye sahip olsa bile farklı bir adresten bağlandığında aynı işlemi yapamaz.

```
public class Startup
{

   option.Cookie = new CookieBuilder {
       HttpOnly = true,
       Name = ".TS.Security.Cookie",
       SameSite = SameSiteMode.Strict
   };
```

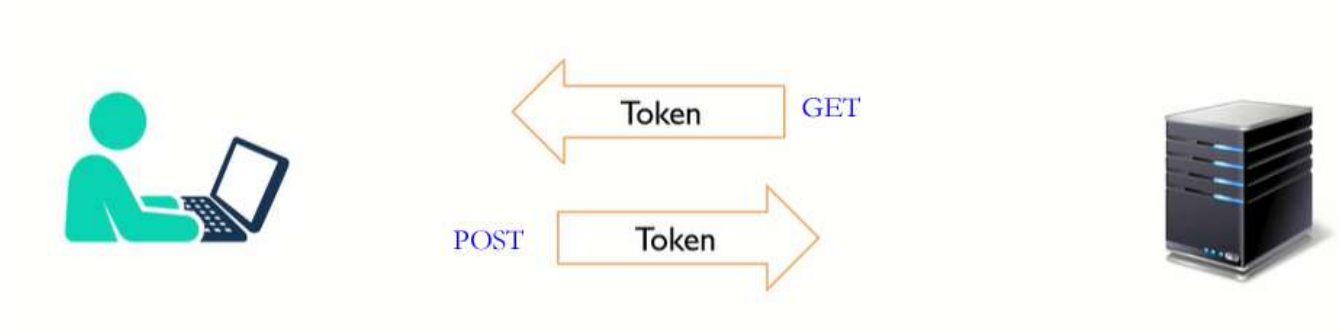
Burada konu olan problem bununda ötesinde. Acaba B kullanıcısı A kullanıcının bilgisayarını kullanarak istediği işlemi yaptırabilir mi? Buna "Cross Side Attack" ları deniyor.

Olayı şöyle bir senaryo ile anlatalım. A kullanıcısı bir bankanın kullanıcısı olarak kendi hesabına girdi. Burada bir havale formunu doldurup bankaya gönderecek. Bu durumda kendi bilgisayarında Cookie, banka bilgisayarında ise Session var demektir. A kullanıcısını onay tuşuna bastığında ödeme bilgileri bankaya gidecektir. Burada kontrol için istenen Cookie bilgisi ve Adres bilgisi tamam demektir. Bu durumda ilk iki kontrolü geçti.

Fakat banka hesabı henüz açıkken, dışarıdan bir hacker kişiye bir mail göndermiş olsa ve bu mailin içerisinde tuzak bir resim olsa ve kişide bu resme tıklasa. Resmin arkasında kişinin göremediği bir form bulunsa ve bu formun içerisinde kişinin hesabından çekilecek bilgiler bulunsa, bu durumda bilgiler bankanın istediği şartları sağlıyor demektir.

Böyle bir senaryoda kişinin bilgisayarında kendisinin açtığı form ile resmin arkasına gizlenen formun banka tarafından birbirinden ayır edilmesi imkansızdır. Bu durumda güvenlik nasıl sağlanacak, şimdi ona bakalım.

Kullanıcı banka hesabından para göndermek üzere bir forma tıklayıp GET metodu ile bu form kullanıcının bilgisayarına gelirken içerisinde bir benzersiz bir şifre yada anahtar (Token) gönderilir. Kişi formu doldurup Submit ettiğinde (POST) işlemi ile formu gönderdiğinde server bu Token bilgisine bakarak aynı formdan bilgilerin gelip gelmediğini kontrol eder.



Şimdi bir form işlemi yapalım bakalım form içinde bu Token oluşacak mı görelim. Kullanıcı giriş yaparken kullandığımız formun kaynak kodlarına bakalım. Formun en son etiketinden önce bu token bilgisini görebiliriz.

Şu ana kadar yaptığımız formlarda Submit işleminde bu token bilgisi server tarafından kontrol edilmiyordu. Güvenlik gerektiren işlemlerde bu kontrolü nasıl yaptırırız şimdi görelim.

Bu işlemi Register sayfasında üye kaydı yaparken güvenlik için Token kontrolünü yaptıralım. Bunun için AccountController içindeki POST türündeki Register metodunun başında aşağıdaki ifadeyi kullanıyoruz. Bunu yazdığımızda sadece Register sayfasından gelen bilgiler için bu kontrolü yapar.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterModel model)
{
```

Şayet controller içindeki tüm metodlar için bu kontrolün yapılmasını istersek bu sefer controller sınıfının en üstüne aşağıdaki ifadeyi yazmalıyız. Böylece her Post metodu için tek tek yazmamış oluruz.

```
namespace TS.WebUI.Controllers
{
    [AutoValidateAntiforgeryToken]
    public class AccountController : Controller
    {
```

Kullanıcı Hesabının Onaylanması

Şu ana kadar kullanıcı hesabını oluşturduk. Kullanıcı giriş yapabiliyor ve giriş yaptığında NavBar üzerindeki linkler değişiyordur. Giriş yapan kullanıcı için güvenlik kontrolü için Cookie ve Session haberleşmelerinden bahsettik. Oysa bir kullanıcının doğru mail adresi vermesi gerekir. Üyeliğinin geçerli olabilmesi için bu kontrolü yaptırmamız gerekir. Dolayısı ile bu derste bu işlemi yapalım.

Startup.cs içinde { `options.SignIn.RequireConfirmedEmail = false;` } şeklinde bir satırımız vardı. Bu satırda geçen false ifadesi kullanıcı hesabını onaylamayı gerekli kılmıyordu. Bunu biz true yapalım ve bir onay işlemi gerekli olsun.

```
options.SignIn.RequireConfirmedEmail = true;
```

Ardından AccountController içine gidelim. Bir asenkron metod oluşturalım. Metodumuzun adı ConfirmEmail olsun. Bu metod dışarıdan iki tane parametre alsın. Bir tanesi kullanıcının Id bilgisi olsun diğeri ise kullanıcının maile göndereceğimiz benzersiz bir sayıyı gösteren Token bilgisi olsun. { `public async Task<IActionResult> ConfirmEmail(string userId, string token)` } . Token bilgisi kullanıcının mailine gidip bu linke tıkladığında kullanıcının hesabı onaylanmış olacak.

Peki biz buradaki metoda dışarıdan gelen bir Token bilgisi alacaksak, bu token ı maile göndermeden önce nerede oluşturacağız. Token bilgisini kullanıcı ilk defa kayıt olduğunda ve kayıt işlemi başarı ile sonuçlandığında oluşturmamız gerekir. Bunun için Post tipindeki Register metodunun içinde kayıt işleminin Succeded olduğu kısım içerisinde maille gönderilecek olan Url adresini oluşturalım. Şu aşamada maile gönderme işlemini yapmayalım. Onu sonraki başlıkta işleyelim. Burada onay işlemini tarayıcının adres satırında çalıştırarak onaylama yapabiliriz.

İlk olarak token için bir "code" oluşturalım. Bu işlemi userManager sınıfı içindeki `GenerateChangeEmailTokenAsync()` metodunu kullanacağız. Bu metod bizden bir user bilgisi alacaktır ve onun içindeki bilgiye göre bir code oluşturup gönderecektir. Tabii oluşturulan token bilgisi veritabanına kaydediliyor ve biz daha sonra veritabanına kaydedilen bu token bilgisi ile onaylama işlemi yapacağız.

```
var code = await _userManager.GenerateChangeEmailTokenAsync(user ,"henüz_mail_adresi_yok");
```

Şimdi token bilgisini oluşturduk (code değişkeni içinde) fakat link oluşturmak için ayrıca userId de gerekir. Bu ikisi kullanılarak Url bilgisini oluşturmalıyız. İlgili UserId nin bilgilerine bakacağız, gönderilen token bilgisi orada varsa ve süresi henüz dolmadıysa kullanıcı onayını gerçekleştireceğiz.

Oluşturacağımız Url bilgisini normal string şeklinde de oluşturabiliriz yada aşağıdaki gibi de Url.Action() metodunu kullanarak oluşturabiliriz. Oluşturulan Url bilgisi içinde linke tıklandığın gidilecek olan controller, action, userId ve token bilgileri olacaktır. Bu işlem için aşağıdaki satırlar tıklanılacak olan url bilgisini oluşturur.

```
var url = Url.Action("ConfirmEmail", "AccountController",
    new { userId = user.Id, token = code });
```

Oluşturulan bu Url bilgisinden sonra kullanıcının mailine gönderme işlemini yapmalıyız. O kısmı henüz şimdi yapmayacağız. Kullanıcının mailine giden url bilgisine tıklama gerçekleşince, url içindeki adres bilgilerine göre AccountController içindeki ConfirmEmail action metoduna gelinecektir. Bu metoda girişte ise userId ve token bilgisi gerekiyordu, bu bilgilerde url içinde bulunmaktadır. Artık metod içinde veritabanındaki kontrol yapılarak kullanıcının onay işlemi tamamlanacaktır.

Biz şu aşamada maile gönderme işlemi yapmayacağımızdan oluşan url bilgisini almak için Console yazdırabiliriz. Buradan aldığımız url bilgisini tarayıcının adres satırında çalıştırırsak onay işlemini gerçekleştirebiliriz.

```
Console.WriteLine(url);
```

Url çalıştırıldığında kullanılacak olan userConfirm() metodunun içini dolduralım. Önce gelen bilgileri kontrol edelim. Eğer userId ve token bilgisi null olarak geldiyse geçersiz bir link gelmiş demektir. Bu durumda kullanıcıya göstermek üzere TempData [""] içerisine bir mesaj yazabiliriz. Yazdığımız bu mesajı sayfada göstermek üzere (_layout.cshtml içinde kullanmıştık) alt yapımız vardı. Yalnız TempData nesnesini kullanabilmek için serilization işlemleri ve gösterim rengini belirlemek içinde CreateMessage isminde bir metod kullanmıştık. Bu metodu AccountController içinde de kullanalım.

```
public void CreateMessage(string message, string alertType)
{
    var msg = new AlertMessage()
    {
```

```
            Message = message,
            AlertType = alertType
        };
        TempData["Message"] = JsonConvert.SerializeObject(msg);
    }



    if(userId==null || token == null)
    {
        CreateMessage("Geçersiz token bilgisi", "warning");
        return View();
    }
```

Ardından gelen userId ye göre böyle bir kullanıcı var mı onun kontrolünü yapalım. Eğer yoksa ona göre kodlarımız aşağıdaki şekilde olacaktır.

```
    var user = await _userManager.FindByIdAsync(userId);
    if(user ==null)
    {
        CreateMessage("Böyle bir kullanıcı yoktur.", "warning");
        return View();
    }
```

Bu satırları pas geçerse böyle bir kullanıcı var demektir. Bu durumda gelen token bilgisi ile kullanıcı içindeki ilgili bilgi karşılaştırılarak onay işlemi yapılır ve geri bir sonuç gönderilir. Buradan dönen sonuç başarılı ise (succeced) kullanıcıya hesabının onaylandığını gösterebiliriz.

```
    var result = await _userManager.ConfirmEmailAsync(user,token);
    if(result.Succeeded)
    {
        CreateMessage("Hesabınız onaylandı.", "success");
        return View();
    }
```

Kullanıcının onaylama işlemi ile ilgili kodlarımız toplu olarak şu şekilde olacaktır.

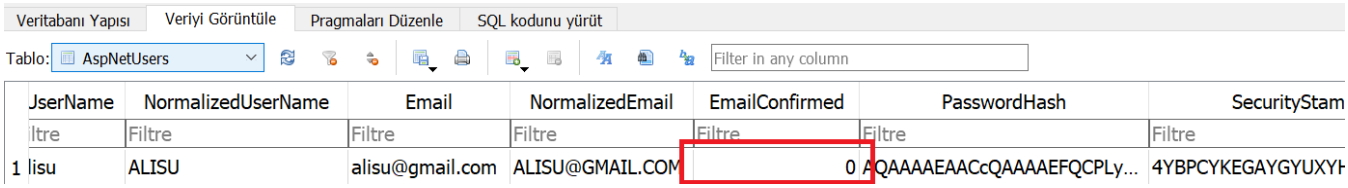
```
public class AccountController : Controller
{
    public async Task<IActionResult> ConfirmEmail(string userId, string token)
    {
        if(userId==null || token == null)
        {
            CreateMessage("Geçersiz token bilgisi", "warning");
            return View();
        }

        var user = await _userManager.FindByIdAsync(userId);
        if(user ==null)
        {
            CreateMessage("Böyle bir kullanıcı yoktur.", "warning");
            return View();
        }

        var result = await _userManager.ConfirmEmailAsync(user,token);
        if(result.Succeeded)
        {
            CreateMessage("Hesabınız onaylandı.", "success");
            return View();
        }

        return View();
    }
```

Şu ana kadar kullanıcının onaylama işlemini yaptık fakat giriş işleminde (Login()) kullanıcının onayı var mı yok mu diye bakmamıştık. Şimdi oradaki düzeltmeleri yapalım.

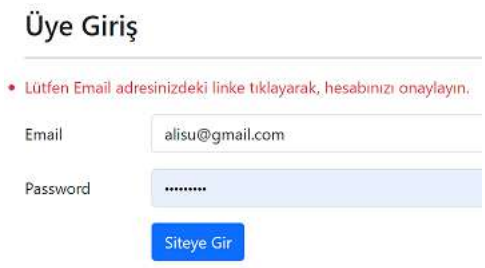
Post tipindeki Login() metodu içerisine yazacağımız aşağıdaki satırlar veritabanında kullanıcının mail alanının onaylı olup olmadığına bakar. Eğer onaylanmamışsa o zaman kullanıcıya ilgili mesajı gösterecektir. Eğer hesabı onaylı ise bu aşamaya takılmayacak ve Login olmuş olur.

```
public async Task<IActionResult> Login(LoginModel model)
{
    if(!await _userManager.IsEmailConfirmedAsync(user))
    {
        ModelState.AddModelError("", "Lütfen Email adresinizdeki linke tıklayarak, hesabınızı
onaylayın.");
        return View(model);
    }
```

Uygulamamızı çalıştırıp deneyelim. Çalıştırmadan önce Veritabanındaki ilgili alanı bir inceleyelim. Dikkat edersek Users tablosundaki EmailConfirmed alanı şu anda 0 yani false olarak gözüküyor. Yani onaylanmamış hesap vardır. Biz alanı onaylı hale getirmeye çalışacağız.



Öncelikle onaylanmadan bir siteye girmeye çalışalım. Bu durumda girişe müsaade etmeyecek ve mail adresimize gitmemizi isteyecektir.

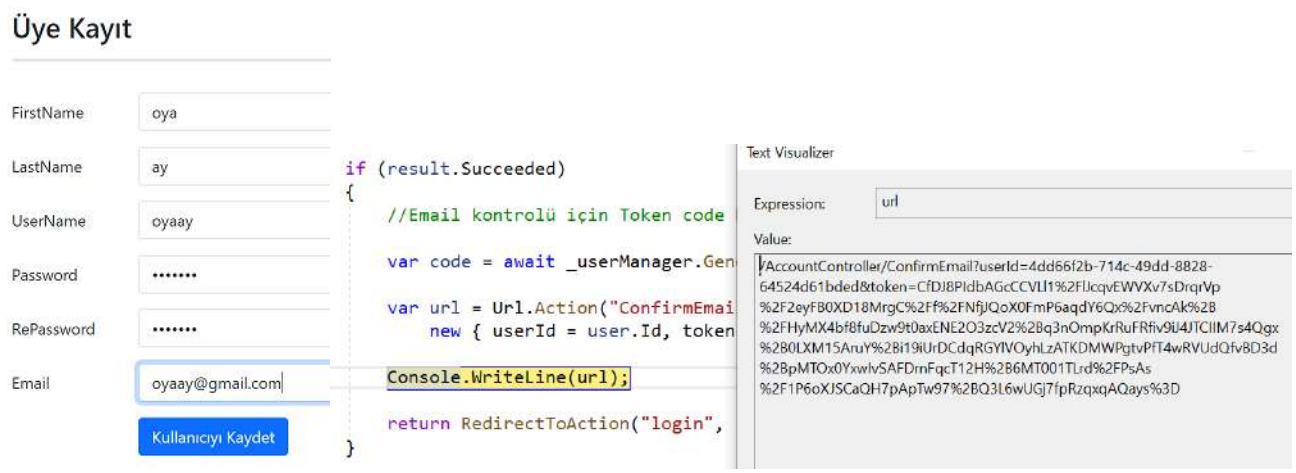


Kişi mailindeki linke tıklayıp hesabını onayladıktan sonra karşısına bir onay sayfasının getirilmesi gerekir. Yani ConfirmEmail() metodunun sonundaki return View(); satırının çalışabilmesi için gideceği bir View sayfası olması gerekir. Şimdi bu sayfayı oluşturalım. Basit bir sayfa olsun. Bu sayfa Views/Account/ConfirmEmail.cshtml şeklinde bir sayfa olacak. Zaten sayfaya _layout üzerinden bir alert kutusu gelecektir. Bu kadar içerik şimdilik yeterlidir.

Views>Account>**ConfirmEmail.cshtml**

```
<h1>Email Onayı</h1>
```

Onaylama için oluşturulacak olan link (url) adresini almamız gerekir. Bunun için yeni bir hesap oluşturalım. Hatırlarsak oluşan linki Consol a yazdırmıştık oradan alalım. Yada programı F9 ile durdurup ilgili url içindeki stringi alalım.



/Account/ConfirmEmail?userId=8089eb62-b79c-4eb9-8066-ce8d791d6ac5&token=CfDJ8PIdbAGcCCVLl1%2FlJcqvEWWTfJV9ECtbJ0djhpuE26r%2BG6PnQxHpOTeX6GWmFmJ3eQrIc5UD89NRjYHUQphZylQvxInsuDcu0gi90iltxdq6XXsiwT72lee%2F%2B%2FU2StL%2BBWGViRcg2KLOlG4Htc13BSokumJbNtzt9ebcWEry4WEI8hBgI1fs1rUEF1IFKDgkGHZ5XYg7pf%2FxAw8zQWuQs5%2BM%2BvgJ1HFP0iB8vZrEEObeOQA%2FKjSFEfg%2FYoRFs7BwevyxFzEbt%2B94xt8%2FpKtBsoA%3D

Bu linki tarayıcımızda localhost adresinin devamına yapıştırırsak Email içinden tıklamış gibi oluruz ve onaylama işlemi gerçekleşir.

{Bu kısımda sürekli token hatası verdi. Sanırım copy-paste esnasında kodda bir hata yapılıyor. Nasıl olsa mail adresine tıklayarak yapılacağından bu aşama geçiliyor. }

Onay Mailinin Kullanıcıya Gönderilmesi

Bir önceki dersimizde kullanıcının tıklayarak onay alacağı Url yapısını oluşturduk fakat bu linkin kullanıcıya gönderilmesi gerekir. Mail yolu ile bu bilginin kullanıcıya gönderilmesi için alt yapının oluşturulması ve Smtp server yoluyla bilginin nasıl ulaştırılacağını bu derste öğrenelim.

Biz kullanıcıya mail gönderirken bir server hizmetini kullanmamız gerekir. Bu hizmeti hosting aldığımız yerin vereceği bir Smtp serverdan alabiliriz yada, mail hizmeti sunan dünya çapındaki firmaların hizmetini kullanabiliriz. Örneğin Gmail, Hotmail gibi firmaların hizmetini kullanabiliriz. Bunların dışında Api kullanabiliriz. Ücretsiz olanlardan bir tanesi için örnek SendGrid dir. Günde 100 adet mail ücretsiz gönderilebilir. Yada ücretli bir Api kullanabiliriz.

Kullanacağımız server için gerekli olan WebUI projesi içine **EmailServices** isminde bir klasör ekleyelim. Bunun içerisine ise bir tane **IEmailSender**.cs adında bir Interface tipinde sınıf ekleyelim (Web projesi üzerine sağ tuşa tıklayıp New Folder>New Item>Interface>IEmailSender.cs sıralamasıyla ekleyebiliriz).

```
namespace TS.WebUI.EmailServices
{
    interface IEmailSender
    {
    }
```

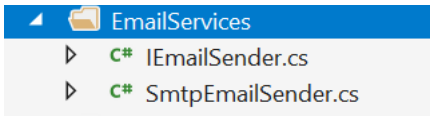
```
    }
```

Biz mail göndermek için IEmailSender interfaceni kullanacağız. Mail göndermek istediğimizde bu interfacesi çağıracağız. İnterfacein arka planınında ona bağlanan Concrete versiyon içinde hangi server kullanılıyor ise onun üzerinden mail gönderme gerçekleşecek.

Interface içerisinde geri dönüşü Task tipinde ve adı `SendEmailAsyc()` tipinde bir asenkron metod tanımlayalım. Bu metod dışırıdan bir mail adresi , konu ve html formatında bir mesaj bilgisi alacak.

```
interface IEmailSender
{
    Task SendEmailAsyc(string email, string subject, string htmlMessage);
```

Şimdi bu IEmailSender ı kullanacak olan bir yapı tanımlayalım. EmailServeces üzerine sağ tuşa tıklayalım yeni bi .cs class yapısı oluşturalım. Class ismini SmtpEmailSender diyelim. Hotmail alt yapısını kullanacağız.



Bu class yapısının içeriğini ise IEmailSender dan türeteceğiz. Bu amaçla sınıf adının yanına şu şekilde yazıyoruz. Ardından ara birimi uygula dediğimizde metod yapıları oluşacaktır.

Sınıf içinde oluşan `SendEmailAsyc()`metodundan önce bir constructer yapısı oluşturalım. Bu yapı dışarıdan gelen server ile alakalı parametreleri buradan alacaktır. Bunun için bir smtp serverın host bilgisi yani bir adresi olmalı (string host). Bir port bilgisi olmalı (int port). Mail şifrelemenin yapılıp yapılmacağını gösteren bir parametre (bool enableSSL), kullanacağımız hotmail adresindeki kullanıcı adı (string username) gerekli. Yani kullandığımız bir mail adresinin username kısmı ve bu hesaba girerken kullandığımız parola bilgisi (string password) gerekli.

```
public SmtpEmailSender(string email, string host, int port, string subject, string htmlMessage,
bool enableSSL, string username, string password)
```

Bunlar dışarıdan gelecek olan parametrelerdir. Concrete sınıfı içinde kullanacağımız parametreler ise bunların aynısı fakat alt çizigili olan versiyonları olacak. Onlarıda daha üst kısımda private tipinde tanımlayalım. Bu değişkenlerin içine dışarıdan gönderilen parametreler içindeki bilgileri constructor yapısı içinde aktaracağız. Bu işlemi yaparken değişkenlerin başında This. İfadesini kullanacağız.

```
private string _email;
private string _host;
private int _port;
private string _subject;
private string _htmlMessage;
private bool _enableSSL;
private string _username;
private string _password;
```

`SendEmailAsyc()` içinde SmptClient sınıfından türetilmiş bir client nesnesi oluşturalım. Bu nesne bizden _host ve _port bilgilerini isteyecektir. Daha sonra bunun içerisinde Credentials özelliğini kullanarak _username ve _password bilgilerini vereceğiz. Ardından EnableSsl özelliği içine ise _enableSSL bilgisini aktaracağız. Bu şekilde Client ımızı (istekde bulunan kişimizi) oluşturuyoruz. Buradaki nesnenin giriş parametrelerinin başında this. ifadesi yani bu class a işaret eden ifade olmalı.

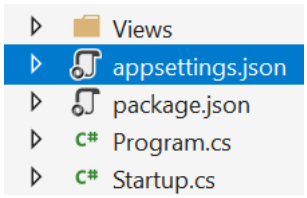
```
var client = new SmtpClient(this._host, this._port)
{
    Credentials = new NetworkCredential(_username, _password),
    EnableSsl = this._enableSSL
};
```

Daha sonra return `client.SendMailAsync()` üzerinden bilgileri göndereceğiz. Bu metod bizden bir mail bekler. Bu mail bilgilerini verirken class ın username ni verdiğimizi söylemek için this._username şeklinde veririz. Diğer

bilgiler ise metod içinde kullandığımız değişkenlerimizdir. Bunların başında this. İfadesinin olmadığına dikkat edelim. Ayrıca mesaj içeriğimizin html formatında olduğunu belirmek içinde `IsBodyHtml = true` olarak bildiriyoruz. Kodlar şu şekilde olur.

```
return client.SendMailAsync(
    new MailMessage(this._username, _email, _subject, _htmlMessage)
    {
        IsBodyHtml = true
    }
);
```

Kullanacağımız Mail Server bilgilerini AppSettings.json içinde vereceğiz. Bunun içinde EmailSender isminde bir özellik tanımlayacağız. Bu özellik içinde hotmail.com un smtp server adresini yazıyoruz. Bu adresin port numaralarını vs öğrenmek için google a "smtp.office365.com smtp settings" yazarak gerekli bilgileri öğrenebiliriz.



Şimdi buradan alınan bilgileri de yazarak kendi hotmail hesabımıza ait bilgileri appsetting.json içine yazalım. Dosyamızın son hali şu şekilde oldu. Tabii burada hotmail hesabı değilde farklı bir hesap kullanırsak o hesaba ait bilgileri yazmamız gerekir. Ayrıca bir hosting kiraladıysak o firmanın verdiği smtp ayarlarını buraya girmemiz yeterlidir.

Appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },

  "EmailSender": {
    "Host": "smtp.office365.com",
    "Port": 587,
    "EnableSSL": true,
    "Username": "<alisu>",
    "Password": "<AliSu-123>"
  },

  "AllowedHosts": "*"
}
```

Bu işlemden sonra **Startup.cs** içinde hangi Interface class ını çağırdığımızda onun dolu versiyonunu kullanacağımızı belirtmek için aşağıdaki satırı içine eklememiz gerekir. Tabii yukarıya namespace adresini eklemeliyiz. Bu satırla IEmailSender interfaceni çağırdığımızda SmtpEmailSender class ı kullanılmış olacak.

```
services.AddScoped<IEmailSender, SmtpEmailSender>();
```

SmtpEmailSender bizden bazı parametreleri bekliyordu. Bunlarıda buradan göndermeliyiz. Buradan göndereceğimiz bilgiler mail server ile alakalı bilgilerdi ve biz bunları Appsetings içerisine yazmıştık. Buradaki

bilgileri Startup içinden ulaşabilmek için startup ın üst kısmına bir Constructer yazalım. Burada kullandığımız IConfiguration nesnesi üzerinden appsettings içindeki bilgilere ulaşacağız.

```
public class Startup
{
    private IConfiguration _configuration;

    public Startup (IConfiguration configuration)
    {
        _configuration = configuration;
    }
```

Configuration nesnesi appsettings içindeki en dıştaki parantez içindeki tüm bilgilere ulaşır. Her bir alt parantez içindeki özellilkerede aşağıdaki şekilde ulaşırız. Startup içinde Configuration ile gelen bilgiler aşağıdaki şekilde aktaracağız.

```
services.AddScoped<IEmailSender, SmtpEmailSender>(i=>
        new SmtpEmailSender(
            _configuration["EmailSender:Host"],
            _configuration.GetValue<int>("EmailSender:Port"),
            _configuration.GetValue<bool>("EmailSender:EnableSSL"),
            _configuration["EmailSender:UserName"],
            _configuration["EmailSender:Password"])
    );
```

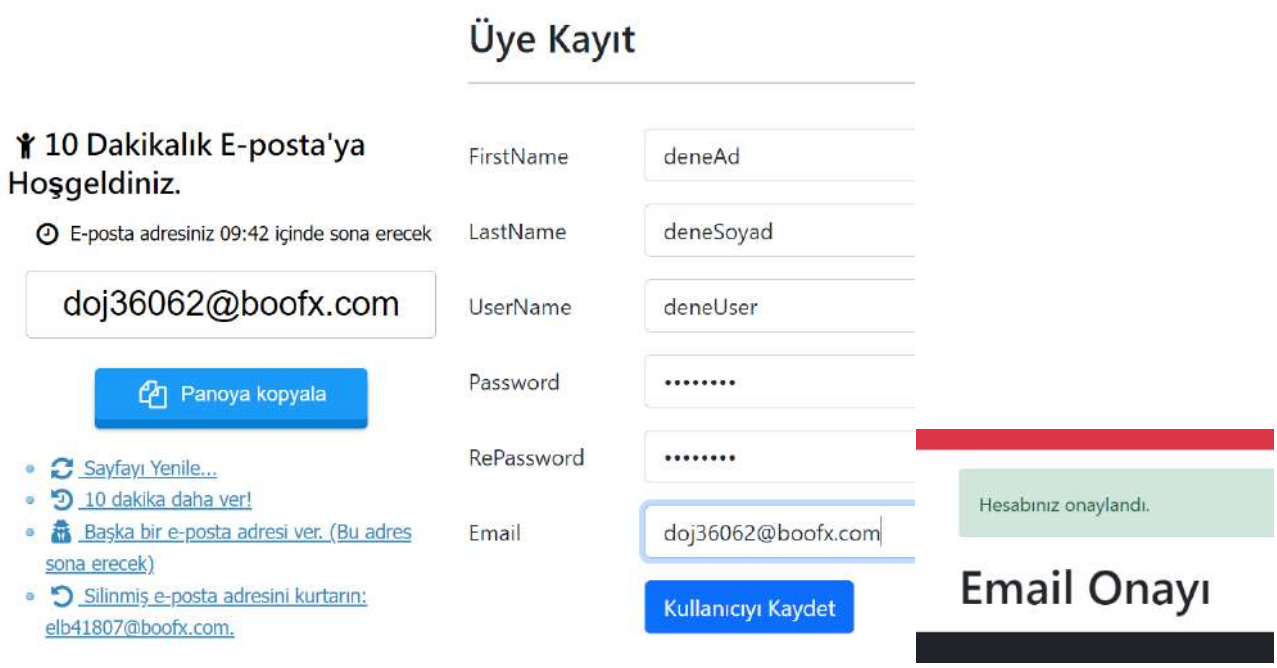
Böylece mail serverın ihtiyaç duyduğu bilgileri Appsettings içinden aldık. Şimdi AccountController içinde maili gönderdiğimiz yere gidelim. Öncelikle AccountController içindeki Constructor kısmına EmailSender nesnesini eklemeliyiz. Dışarıdan EmailSender gönderilecek bunu içeri alabilmeliyiz.

```
public class AccountController: Controller
{
    private UserManager<User> _userManager;
    private SignInManager<User> _signInManager;
    private IEmailSender _emailSender;
    public AccountController(UserManager<User> userManager, SignInManager<User>
signInManager, IEmailSender emailSender)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
    }
```

Controller içinde aşağılarda mail gönderdiğimiz Register() metodunun içinde url bilgisini oluşturduğumuz yere gelelim ve mail gönderme işlemi için kodları aşağıdaki şekilde yazalım. Burada üç tane bilgi gitmektedir. Email adresi, Subject ve Message kısmıdır. Mesaj kısmı içinde tıklanacak linkin nasıl verildiğine dikkat edelim. Mesaj içeriği http şeklinde olduğundan başına $ işareti konuldu. url adresinin domain name sonuna nasıl eklendiği de önemlidir.

```
await _emailSender.SendEmailAsyc(model.Email,"Mail onaylama",$"Lütfen mailinizi onaylamak için
linke <a href='https://localhost:44310{url}'>tıklayınız</a>");
```

Kodlarımızı deneyelim. Bunun için 10 dakikalık geçici bir mail adresi alalım. Google “10 dakikalık mail adresi” yazalım ve gelen ekranda bize sunulan geçici mail adresini alalım.



Kodların Son Hali Gerekli düzelmeler yapıldı

IEmailSender.cs

```
using System.Threading.Tasks;

namespace shopapp.webui.EmailServices
{
    public interface IEmailSender
    {
        Task SendEmailAsync(string email, string subject, string htmlMessage);

    }
}
```

SmtpEmailSender.cs

```
using System.Net;
using System.Net.Mail;
using System.Threading.Tasks;

namespace shopapp.webui.EmailServices
{
    public class SmtpEmailSender : IEmailSender
    {
        private string _host;
        private int _port;
        private bool _enableSSL;
        private string _username;
        private string _password;
        public SmtpEmailSender(string host, int port, bool enableSSL, string username, string
password)
        {
            this._host = host;
            this._port = port;
            this._enableSSL = enableSSL;
            this._username = username;
            this._password = password;
        }
        public Task SendEmailAsync(string email, string subject, string htmlMessage)
```

```
        {
            var client = new SmtpClient(this._host, this._port)
            {
                Credentials = new NetworkCredential(_username, _password),
                EnableSsl = this._enableSSL
            };

            return client.SendMailAsync(
                new MailMessage(this._username, email, subject, htmlMessage)
                {
                    IsBodyHtml = true
                }
            );
        }
    }
}
```

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },

  "EmailSender": {
    "Host": "smtp-mail.outlook.com",
    "Port": 587,
    "EnableSSL": true,
    "UserName": "icayiroglu@karabuk.edu.tr",
    "Password": "Sifre-123"
  },

  "AllowedHosts": "*"
}
```

Startup.cs (sadece ilgili yerler bırakıldı)

```
using shopapp.webui.EmailServices;

namespace TS.WebUI
{
    public class Startup
    {
        private IConfiguration _configuration;
        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }

            services.AddScoped<IEmailSender, SmtpEmailSender>(i=>
                    new SmtpEmailSender(
                        _configuration["EmailSender:Host"],
                        _configuration.GetValue<int>("EmailSender:Port"),
                        _configuration.GetValue<bool>("EmailSender:EnableSSL"),
                        _configuration["EmailSender:UserName"],
                        _configuration["EmailSender:Password"])
                );
}
```

AccountController.cs

```
using shopapp.webui.EmailServices;

namespace TS.WebUI.Controllers
{
    public class AccountController: Controller
    {
        private IConfiguration _configuration;
        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }

        //****************** REGISTER **********************
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Register(RegisterModel model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = new User()
            {
                FirstName = model.FirstName,
                LastName = model.LastName,
                UserName = model.UserName,
                Email = model.Email
            };

            var result = await _userManager.CreateAsync(user, model.Password);

            if (result.Succeeded)
            {
                //Email kontrolü için Token code bilgisi oluşturuluyor

                var code = await
_userManager.GenerateChangeEmailTokenAsync(user,"henüz_mail_adresi_yok");

                var url = Url.Action("ConfirmEmail", "Account",
                    new { userId = user.Id, token = code });

                await _emailSender.SendEmailAsync(model.Email,"Mail onaylama",$"Lütfen
mailinizi onaylamak için linke <a href='https://localhost:44310{url}'>tıklayınız</a>");

                return RedirectToAction("login", "account");
            }

            ModelState.AddModelError("", "Şifre kurallara uygun değil yada kullanıcı ve Email
daha önceden kayıtlı olabilir");

            return View(model);
        }
}
```

Şifremi Unuttum Uygulaması

Şifremi Unuttum Sayfası (Forgot Password)

Bu uygulamada bir önceki uygulamada yaptığımıza benzer bir çalışma yapacağız. Kişi şifresini unuttuğunda mail adresine link içinde bir kod göndereceğiz. Bu linke tıkladığında şifresini değiştireceği sayfaya yönlendirip yeni şifresini girmesini sağlayacağız.

Öncelikle AccountController içinsde ForgotPassword uygulaması için bir Action metod oluşturalım. Bunun get ve post versiyonlarının ikisinide oluşturalım. Metodun tipi yazılmazsa Get metodu olduğu anlaşılır. Get metodu Form sayfasına giderken çalışacak olan metoddur.

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

Şimdi View sayfamızı oluşturalım. Sayfamızın alt yapısını Login.cshtml sayfasınadn alalım. Sayfada model kullanmayacağımızdan ilgili alanlar içindeki Asp-for ifadelerini çıkaralım. Bir textbox ile bir tane buton olacak..

ForgotPassword.cshtml

```
<h1 class="h3">Şifre Yenileme</h1>
<hr>
<div class="row">
    <div class="col-md-12">
        <div asp-validation-summary="All" class="row text-danger"></div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <form asp-controller="Account" asp-action="ForgotPassword" method="POST">
            <div class="form-group row mb-3">
                <label class="col-sm-2 col-form-label">Email</label>
                <div class="col-sm-10">
                    <input class="form-control" type="email" name="Email">
                </div>
            </div>

            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2">
                    <button type="submit" class="btn btn-primary">Gönder</button>
                </div>
            </div>

        </form>
    </div>
</div>
```

Post metodu ise formdan gelen bilgileri işleyecek olan metoddur. Dolayısı ile form sayfamızdan (View sayfamızdan) bir mail adresi gelecek. Dolayısı ile Model kullanmak yerine basit bir string parametre ile email bilgisini taşıyalım. Post metodumuz Asenkron Task tipinde olacak. View sayfasından gelen Email bilgisi boş olursa tekrar sayfaya yönlendirelim. Gelen Email bilgisi boş değilse bu maille kayıtlı bir kullanıcı var mı onu kontrol edelim. Eğer bu mail adresi ile kayıtlı bir user bilgisi yoksa kullanıcıyı tekrar sayfaya gönderebiliriz. Bu gibi durumlarda kullanıcıyı bilgilendirmek için mesajları da göndermek gerekir. Sonraki satırlarda için eğer bir kullanıcı varsa mailine göndermek üzere bir token bilgisi üretelim. Fakat burada üretilen token Reset işlemi için olduğundan farklı bir nesne kullanarak üretelim. Bu metodlar kullanılırken başına await ifadesi kullanılır. İşlem bitene kadar beklemesi için yazılır. Oluşturulan token bilgisinden sonra kullanıcıya bir mail gönderelim. Bu aşamaları daha önce yapmıştık. Benzer işlemler olacak. Linke tıklandıktan sonra gidecek sayfa ise ResetPassword sayfası olacaktır. Bununla ilgili action metoduna yönlendirme yapılacak. Anlatılan Post metodunun kodları aşağıdaki şekilde olur.

```
    [HttpPost]
    public async Task<IActionResult> ForgotPassword(string Email)
    {
        if(Email==null)
        {
            //Burada mail adresini girmesi için kullanıcıyı bilgilendirmek gerekir. Mesaj
gönderilmeli.
            return View();
        }

        var user = await _userManager.FindByEmailAsync(Email);
        if(user==null)
        {
            //Burada böyle bir kullanıcı olmadığına dair mesajıda vermek gerekir.
            return View();
        }

        var code = await _userManager.GeneratePasswordResetTokenAsync(user);
        var url = Url.Action("ResetPassword", "Account", new
        {
            userId = user.Id,
            token = code
        });

        // email gönderme
        await _emailSender.SendEmailAsync(Email, "Şifre Resetleme", $"Lütfen şifrenizi
resetlemek için linke <a href='https://localhost:44310{url}'>tıklayınız.</a>");

        return View();
    }
```

Şifre Resetleme Sayfası (Reset Password)

Linke tıklanıp geri döndükten sonra Resetleme sayfasına gidecek. Buradan kişinin yeni şifresi alınıp veritabanından güncelleme işlemleri yapılacak. Bununla ilgili ResetPassword() metodlarını (Get ve Post) yazalım.

Get metoduna kişinin mailinde tıkladığı link üzerinden gelmiştik. Dolayısı ile link içerisinde sakladığımız bilgiler ise userId ve token bilgileridir. Bu bilgilerin url içinde çekilmesi ve Get metoduna giriş parametreleri olarak kullanılması gerekecek. Ardından bu metod sayesinde bu prametreler ResetPassword.cshmtl sayfasına götürülmesi gerekir. O sayfada ise bilgiler form içindeki bir hidden alanında tutulmalıdır. Çünkü bunlar ordan dönüş metodu olan Post a geldiğimizde veritabanında kullanacağımız bilgilerdir. Dönüş yolunda post metoduna gelirken 3 tane bilgiyi getirmemiz gerekir. Kişiyi tanımladığımız userId ve reset işlemini kontrol eden token bilgisi ve yeni şifresinin olduğu password bilgisi. Bu üç bilgiyi form içinden alıp post metoduna getirken model yapısı içinde taşıyalım. Dolayısı ile View sayfasında ResetPasswordModel isminde bir model sınıfını kullanalım. Kodlar aşağıdaki şekilde olur. Formda [Required] kısımlar doldurulmasını zorunlu kılar. DataType.EmailAddress kısmı yazım formatının email formatında olup olmadığını kontrol eder. Datatype.Password kısmı ise şifreleri yazarken göstermemek içindir.

Models> **ResetPasswordModel.cs**

```
using System.ComponentModel.DataAnnotations;

namespace TS.WebUI.Models
{
    public class ResetPasswordModel
    {
        [Required]
        public string Token { get; set; }
        [Required]
        [DataType(DataType.EmailAddress)]
```

```
        public string Email { get; set; }
        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

View>Account>ResetPassword.cshmtl sayfamızı oluşturalım. İçeriğini Login sayfasında kopyalarak düzenleyelim. Sayfamızda model yapısı kullanılacağından alanlar için kullandığımız name parametereler gerek yoktur. Asp-for parametreleri gerekli biglileri model içinden otomatik olarak alacaktır. Sayfa içerisinde 3 tane bilgi tutacağız. Mail adresi, Şifre ve Token bilgisi olacaktır. Token bilgisi kullanıcıya gösterilmeyeceğinden hidden tipinde olacaktır.

ResetPassword.cshtml

```
@model ResetPasswordModel

<h1 class="h3">Şifre Yenileme</h1>
<hr>
<div class="row">
    <div class="col-md-12">
        <div asp-validation-summary="All" class="row text-danger"></div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <form asp-controller="Account" asp-action="ResetPassword" method="POST">

            <input type="hidden" asp-for="Token" />

            <div class="form-group row mb-3">
                <label asp-for="Email" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Email">
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="Password" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Password">
                    <span asp-validation-for="Password" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2">
                    <button type="submit" class="btn btn-primary">Kaydet</button>
                </div>
            </div>
        </form>
    </div>
</div>
```

Bu sayfaya gidecek olan Get tipindeki ResetPassword() action metodunu oluşturalım. Kullanıcı mail adresine gelen linke tıkladığında bu link bu metoda yönlendirecektir. Link içerisinde bulunan UserId ve Token bilgileri bu metoda girişte alınmalı. Metod içerisinde ise gelen bilgilerin doğruluğunun kontrolünün yapılması gerekir. Eğer gelen userId yada token bilgisi gelmediyse ana sayfaya yada bir hata sayfasına yönlendirilebilir. Bu bilgiler null değilse View sayfasına götürmek üzere model içerisine token bilgisini atabiliriz. Get tipindeki metodumuz şu şekilde olur.

```
    public IActionResult ResetPassword(string userId, string token)
    {
        if(userId==null || token ==null)
        {
            //Burası bir hata sayfasına yönlendirip gelen bilgilerin hatalı olduğu
yazdırılabilir.
            RedirectToAction("Home", "Index");
        }

        var model = new ResetPasswordModel { Token = token };

        return View();
    }
```

Post tipindeki metodumuzu oluşturalım. View sayfasından model içinde bilgilerimiz gelecektir. Üç adet bilgi gelecektir. Token bilgisi, kişinin eliyle formda doldurduğu mail ve şifre bilgileri olacaktır. Öncelikle gelen metod içindeki bilgilerin Validation doğrulamasını kontrol edelim. Eğer gelen validation geçersiz ise tekrar view sayfasına gidelim ve getirdiğimiz modeli yanımızda götürelim.

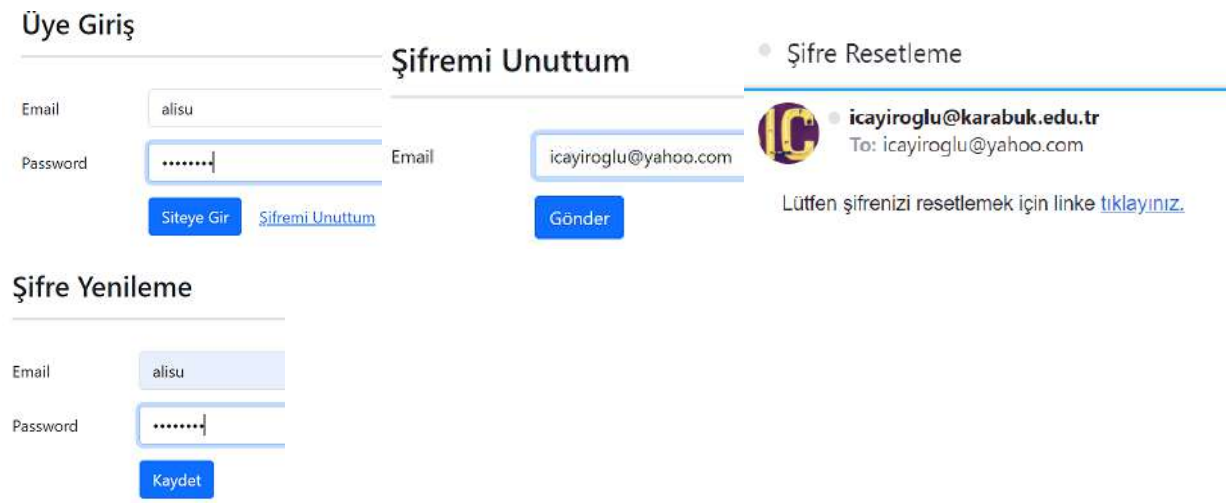
Ardından gelen mail adresi ile bir kullanıcı var mı onun kontrolünü yapalım. User varsa artık şifre resetlemeyi yapabiliriz. Resetleme gerçekleşirse ve sonuç true dönerse artık işlem gerçekleşmiş ve kişinin yeni şifre ile siteye giriş yapması için Login sayfasına yönlendirebiliriz. Eğer bunlar haricinde bir hata varsa en son işlemde tekrar sayfaya yönlendirilebilir. Giderkende yanımızda model bilgisini götürmeliyiz.

```
    [HttpPost]
    public async Task<IActionResult> ResetPasswordAsync(ResetPasswordModel model)
    {
        if(!ModelState.IsValid)
        {
            return View(model);
        }

        var user = await _userManager.FindByEmailAsync(model.Email);

        if(user==null)
        {
            //Burada ana sayfaya gitmek yerine bu maile ait bir kullanıcı bulunmadığına dair bir
hata mesajına gitmesi daha uygundur.
            return RedirectToAction("Index","Home");
        }

        var result = await _userManager.ResetPasswordAsync(user,model.Token,model.Password);
        if(result.Succeeded)
        {
            return RedirectToAction("Login","Account");
        }

        return View(model);
    }
```

Son olarak Login sayfası için şifremi unuttum linkini koyalım. Kişi giriş yapmaya çalıştığında şifresini hatırlamayazsa bu link üzerinden şifresini resetleyebilsin.

```
    <div class="form-group row mb-3">
        <div class="col-sm-10 offset-sm-2">
            <button type="submit" class="btn btn-primary">Siteye Gir</button>
            <a href="/account/forgotpassword" class="btn btn-link" >Şifremi Unuttum</a>
        </div>
    </div>
```

Kodları denediğimizde çalıştığını görürüz.



Mesaj Yapısının Extension Metod (ekstradan yazılmış) Şeklinde Düzenlenmesi

Şu ana kadar kullanıcıya gösterdiğimiz Alert mesajları iki şekilde gösterdik. TempData ve motel.State yöntemlerini kullandık. Fakat bu işlemleri Extension denilen (genişletme) kendi içinde oluşturulan bir sınıf yapısı ile yaparsak çok daha kullanışlı hale getirmiş oluruz. Bu şekilde bir sınıf yapısı ve TempData alt yapısını kullanarak mesaj yöntemimizi daha kullanışlı hale getirelim.

Burada extension metodu TemData nın sonuna bir nokta koyup onun alt metodlarını çalıştırmak üzerine bir yöntemdir. Bu yöntem bir string içinde yazılabilir. Örneğin bir string metnin içerisinden boşluk karakterleri çıkarmak istiyoruz. İşte bu sitring sonuna bir nokta koyup hangi extension yazdıysak onu çalıştırıp boşlukları çıkarma işlemini yaptırabiliriz. Tabii yazacağımız metod üst bir sınıfın alt metodu olarak yazılacaktır. String bir sınıftır, TempData bir sınıftır. Bunların içerisine alt bir metod olarak yazacağız. Yani burada Extension dediğimiz metod sınıfın içine yazılmış ekstradan bir metod ve bizim mevcut metodların içine eklediğimiz kendimize ait bir metoddur.

Biz burada uygulama olarak TempData içine yazalım ve Alert mesaj uygulamasında bu metodu kullanalım.

İlk olarak daha önceden yazdığımız Models>AlertMessage.cs içindeki sınıf yapımızı düzenleyelim. Mesaj içerisinde bir başlık, mesaj ve mesaj tipi olsun. Yani alert mesajı verirken üç bilgiyi kullanalım.

Models> **AlertMessage.cs**

```
namespace TS.WebUI.Models
{
    public class AlertMessage
    {
        public string Title { get; set; }
        public string Message { get; set; }
        public string AlertType { get; set; }
    }
}
```

Alert mesajı gösterdiğimiz html kodlarını bir partial view içine alalım. Her zaman konrolü kolaylaştırmak için partial view kullanılması uygundur. Bunun için Shared klasörü içinde _ResultMessage.cshmtl şeklinde bir partial sayfa oluşturalım. İçerisine _Layout.cshtml içinden alacağımız mesaj bölümündeki kodları yapıştıralım. ResultMessage sayfasının modeli olarak yukarıda oluşturduğumuz model yapısını kullanalım. Bu sayfanın son hali aşağıdaki şekilde olur. Kendisine model içinde gelen bilgileri işleyerek sayfada görüntüleyecektir.

Views>Shared>**_resultMessage.cshtml**

```
@model AlertMessage
<div class="row">
    <div class="col-md-12">
        <div class="alert alert-@Model.AlertType">
            <h4 class="alert-heading">@Model.Title</h4>
            <p>@Model.Message</p>
        </div>
    </div>
</div>
```

Bu partial view kullanacak _Layout sayfası içindeki kısmı düzenleyelim. Bu kısımda partial sayfasına linkimiz vardır. İçerisinde kullandığımız mesajın serilization kodlarını extension metod içine alacağız. Burada geçen model parametresini daha sonra düzenleyeceğiz.

```
    <main class="mt-3">
        <div class="container">
            @if (TempData["Message"] != null)
            {
               <partial name="_resultMessage" model=""/>
            }
            @RenderBody()
        </div>
    </main>
```

Şimdi TemData için yazmak istediğimiz Extension Metodu yazalım. Extension metodların hepsini bir klasör içinde toplayalım. Bunun için WebUI projesi içinde Extensions isminde bir klasör oluşturalım. Ve İçerisinde TempDataExtensions sınıf dosyasını oluşturarak içini dolduralım. Oluşturacağımız bu sınıf static tipinde olacak. Ve bunun içerisine ekleyeceğimiz metodlarda static tipinde olacak. Bunun içerisine bir tane Put metodu olacak. Yani TempData'nın içerisine bilgi atarken serilize edip atma işini burada yapacağız. Bir tane de Get metodu olacak. Yine içerisindeki bilgileri diserilize ederek çözeceğiz. Serilize edeceğimiz bilginin tipi de T olacak. Yani generic bir ifade olacak. Bu put ve get metodunu hangi tip ile çağıracağımızı This. İfadesiyle de belirtiyoruz. Yani kullanacağımız TempDataların tipi ItempDataDictionary şeklindedir. Bir TempData üzerine mouse ile geldiğimizde bu tipte oluşturulduğunu görebiliriz. Bu tipi Put metodunun sonuna ekliyoruz. Ctrl (.) ile namespaceni yukarıya eklememiz de gerekir. TempData içerisine bilgi atarken bir tane de key bilgisine ihtiyamız var. Ayrıca bir tane Value bilgisine ihtiyacımız olacak. Bu value bilgisi ise T tipinde olacak. Yani generic değişebilen bir tipde olacak. En sonda ise T nin bir class olacağını da söyleyebiliriz. Sonuct biz TempData içine bir class atmaya çalışıyoruz. Ve bunun içinde T nin bir class olacağını belirtmemizde fayda vardır.

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Newtonsoft.Json;

namespace TS.WebUI.Extensions
{
    public static class TempDataExtensions
    {
    }
}
```

Şimdi put metodunun içini dolduralım. Gönderdiğimiz T sınıf tipindeki bilgiyi Json kütüphenesini kullanarak serilize edelim ve çıkan sonucuda tempData nın key bilgisi içine atalım. Biz burda tempData nın içerisine bilgi atma işlemini gerçekleştirmiş oluyoruz.

```
    public static void Put<T>(this ITempDataDictionary tempData, string key, T value)where
T:class
    {
        tempData[key] = JsonConvert.SerializeObject(value);
    }
```

İkinci metodumuzda ise serilize ettiğimiz bilgiyi Deserilize ederek geri almamız gerekiyor . Bu bilgiyi alırken T value ifadesi olmaz. Key bilgisini kullanacağız. TempData üzerinde taşıdığımız key bilgisini kullanacağız. Ama ortaya çıkan

bilgi T tipinde olacağından sondaki "where T:class" ifadesi kalabilir. Bu ifadeyi yazmazsak class dan farklı tipte bir T bilgisi olursa uyarı vermesi için kullandık. Burada kullandığımız Get metodunda geriye bir T döndürmemiz gerekiyor. TempData içinden key bilgisini TryGetData() metodu ile alabiliriz. Aldığımız bilgiyi de object tipindeki o içine atabiliriz. "o" bilgisini geriye göndermeliyiz. Ama geriye gönderirken bir null olma durumu var mı kontrol edelim. Çünkü bir key bilgisi yada value bilgisi gelmezse geriye null döner. Peki o bilgisi eğer null ise geriye null gönderelim, null değilse bu sefer Deserilize edelim ve T nin içine atalım. T de geriye dönen sınıf olacaktır. Deserilize etmeden önce o bilgisini string ifadeye çevirelim. Bu kısım biraz karışıktır. Açılan kapanan parantezlere dikkat etmek gerekir. Yani deserilize etmek için o bilgisini vermemiz gerekiyor. Fakat o bilgisi bir object tipindedir. Dolayısı ile önce stringe çeviriyoruz. Daha sonra bunu deserilize edip T nin içine atıyor.

Artık extension metodumuz hazır tüm kodları toptan verirsek TempDataExtensions class ı şu şekilde olmuş olur.

WebUI>Extensions> **TempDataExtensions.cs**

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Newtonsoft.Json;

namespace TS.WebUI.Extensions
{
    public static class TempDataExtensions
    {
        public static void Put<T>(this ITempDataDictionary tempData, string key, T value)where
T:class
        {
            tempData[key] = JsonConvert.SerializeObject(value);
        }
        public static T Get<T>(this ITempDataDictionary tempData, string key) where T : class
        {
            object o;
            tempData.TryGetValue(key, out o);
            return (o == null ? null : JsonConvert.DeserializeObject<T>((string)o));
        }
    }
}
```

Oluşturduğumuz bu sınıfın View sayfalarında çalışması için namespace ni (`TS.WebUI.Extensions`) ViewImport içerisine eklemeliyiz.

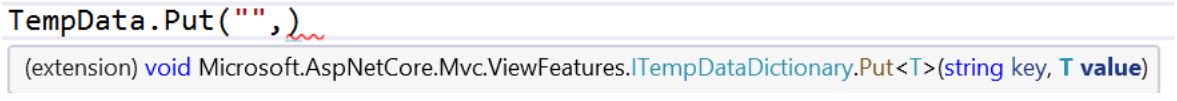
```
@using TS.WebUI.Extensions
```

Artık AccountController içinde bu işler için kullandığımız CreateMessage() metodunu silelim. Bu mesajı nerede kullandıysak oraları yeni metodumıza göre düzenleyelim.

Artık TempData içerisine bir bilgi atarken TempData.Put() kullanabiliriz. Controller sınıfın en yukarısına namespace ekleyelim.

```
using TS.WebUI.Extensions;
```

Aşağılarda artık TempData içindeki bizim eklediğimiz Put() metodunu kullanabiliriz. Bizden iki tane parametre isteyecektir. String tipinde key bilgisi, T tipinde value bilgisi olacaktır.



Burada kullanılan key bilgisi aslında bir başlığı ifade ediyor. O yüzden buraya "message" ismini verelim. Value değeri ise bir sınıf şeklinde olacak (T bir sınıfı ifade ediyordu) ve içerisinde mesaj bilgileri olmalı. Dolayısıyle bu value değerini AlertMessage.cs modelimizden türeteceğiz. Buna göre burasını şu şekilde yazmalıyız.

```
    TempData.Put("message", new AlertMessage()
    {
        Title="Token Hatası",
        Message ="Geçersiz token bilgisi",
        AlertType ="danger"
    });
```

Bu kullanımı CreateMessage kullandığımız her yere yazalım.

```
    TempData.Put("message", new AlertMessage()
    {
        Title = "Hesap sonucu",
        Message = "Hesabınız onaylandı.",
        AlertType = "success"
    });
```

AdminController içinde de CreateMessage metodu kullanmıştık. Oradaki mesaj ifadelerini de değiştirelim.

Şimdi yukarıda yarım bıraktığımız bir yer vardı. Orayı tamamlayalım. **_layout** içisinden resultMessage.cshtml ye giderken yanımızda model bilgisini götürmemiz gerekiyordu. Bu kısmı aşağıdaki gibi yazabiliriz.

```
   <main class="mt-3">
        <div class="container">
            @if (TempData["Message"] != null)
            {
                <partial name="_ResultMessage" model="@(TempData.Get<AlertMessage>("message"))"
/>
            }
            @RenderBody()
        </div>
    </main>
```

Kodları deneyelim. Bir kullanıcıdan çıkış yaptığımızda Logout() metodu içine aşağıdaki gibi bir kod ekleyelim.

```
   public async Task<IActionResult> Logout()
    {
        await _signInManager.SignOutAsync();

        TempData.Put("message", new AlertMessage()
        {
            Title = "İşlem Sonucu",
            Message = "Oturumunuz Kapandı",
            AlertType = "success"
        });
        return Redirect("~/");
    }
```

Bu işleme göre bir çıkış yaparsak aşağıdaki sonucu alırız.

ÜYELİK YÖNETİMİ

Rollerin Eklenmesi

Şu ana kadar yaptığımız uygulamalar şu şekilde çalışmaktadır. Örneğin admin kontrolü yaparken AdminController başına `[Authorize]` kelimesini eklemiştik. Bu kelime kişinin mutlaka login olmasını mecbur bırakıyordu. Kişi login olduğunda ise onun bilgisayarına bir Cookie bırakıyorduk ve sonraki ziyaretlerinde oradaki cookie nin kontrolünü

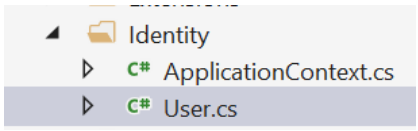
yaparak girişine müsaade ediyorduk. Böyle bir uygulamda tarayıcısında cookinin bulunduğu herkes bu yetkiye sahip olmuş oluyor. Yani kullanıcı girişi yapan kişi müşteride (customer) olabilir admin de olabilir. Böyle bir durumda müşteri admin için oluşturulan sayfaları görebilir.

Bu nedenle kullanıcı oluşturmanın yanında kullanıcının rolleride önemlidir. Belli sayfalara belli rollere sahip kişiler ulaşabilmelidir. Örneğin AdminController altındaki metodlara sadece Admin rolüne sahip kişiler ulaşabilmelidir. Yani controller başındaki ifade şu şekilde olmalıdır.

```
[Authorize(Roles="Admin")]
```

Şimdi rol işlemleri için gerekli alt yapıyı oluşturmaya başlayalım. UserManager sınıfı kullanıcı oluşturmak ile alakalı işlemleri yapıyordur. Roller ile alakalı işlemleri ise RoleManager sınıfı yapmaktadır. AdminController içinde yapacaklarımızı yazalım.

Controller ın en yukarısında Private tanımlamamızı yapalım. Öncelikle user işlemleri için ne yaptığımızı bir hatırlayalım. Daha önceden user sınıfını oluşturmak için Identity klasörü içinde bir User sınıfı oluşturmuştuk. Bizim oluşturduğumuz bu user sınıfı asp yi hazırlayanların daha önceden yazdığı IdentityUser sınıfından türetiliyordu. Biz oradaki alanların üzerine ekstradan kendi eklediğimiz FirstName ve LastName alanları koymuştuk. Böylece daha az tanımlama ile uğraşarak tüm User için gerekli alanları oluşturmuş oluyorduk.




```
public class User:IdentityUser
{
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }
```

Benzer işlemi Roller içinde yapabiliriz. Yani asp içinde bulunan IdentityRole tablosunda bulunan özelliklerin üzerine ekstra özellikler ekleyerek yeni bir sınıfı oluşturabiliriz. Fakat burada bunu yapmayacağız, IdentityRole sınıfının içine konulan temel özellikleri kullanacağız. Dolayısı ile kendimiz bir sınıf oluşturmamıza gerek kalmayacak. Direk AdminController içinde bu sınıfı kullanabiliriz. Tanımlamayı aşağıdaki şekilde yapalım.

```
private RoleManager<IdentityRole> _roleManager;
```

Bu satırların bize anlattığı durum role işlemleri için RoleManager sınıfını kullanacağız. Bu sınıfı içinde IdentityRole sınıfı içinde bulunan özellikleri kullanacağız. Bu işlemleri yaparken Admincontroller içinde kullanacağımız sınıf değişkenimiz ise _roleManager olacaktır. Yani bizim kullanacağımız _roleManager ifadesi asp nin hazır olan RoleManager kütüphanesinin özelliklerini taşıyacak ve onun yaptığı işlemleri yapacak. Bunların dışında bir de AdminController sınıfına dışarıdan gelen Role özellikleri olacak. Bu özellikleri burada kullanacağımız _roleManager içine aktarmamız gerekiyor. Bu işlem ise injection işlemi diyeceğiz ve bu işlemi şu şekilde yapıyoruz. Ayrıca adminController içinde user bilgilerinide ihtiyacımız olacak. Benzer tanımlamaları onun içinde yapalım.

```
    private RoleManager<IdentityRole> _roleManager;
    private UserManager<User> _userManager;

    public AdminController (IProductService productService,
                            ICategoryService categoryService,
                            RoleManager<IdentityRole> roleManager,
                            UserManager<User> userManager
                            )
    {
        _productService = productService;
        _categoryService = categoryService;
        _roleManager = roleManager;
        _userManager = userManager;
    }
```

Admincontroller içinde kullanacağımız metodlarımızı yazalım. Rollerin listesini getirmek üzere Get tipinde RoleList() metodu yazalım. Ayrıca Role eklemek içinde RoleCreate() metodunu yazalım. Listeleme metodunda sayfadan geri dönen bir işlem olmayacağından bunun Post versiyonuna ihtiyaç yoktur. Ama RoleCreate() işleminde sayfada bir kayıt gerçekleşeceğinden bunun Post versiyonu oluşturulacak. Sayfadan buraya sadece role Name

gelse yeterlidir. İstersek burası içinde bir model tanımlayabiliriz. Biz içerisinde tek bir değişken olan basit bir model tanımlayarak yapalım.

```
    public IActionResult RoleList()
    {
        return View();
    }
    public IActionResult RoleCreate()
    {
        return View();
    }
    [HttpPost]
    public IActionResult RoleCreate(RoleModel model)
    {
        return View();
    }
```

Models klasörü altındaki Model yapımız şu şekilde olur

Models>**RoleModel.cs**

```
using System.ComponentModel.DataAnnotations;

namespace TS.WebUI.Models
{
    public class RoleModel
    {
        [Required(ErrorMessage = "Role ismi girilmelidir")]
        public string Name { get; set; }
    }
}
```

Views>Admin> altında RoleList.cshtml ve RoleCreate.cshmtl sayfalarını oluşturalım. Create sayfası içinde bir form olacak. Örnek formu CategoryCreate sayfasından alıp düzenleyelim.

Views>Admin>**RoleCreate.cshtml**

```
@model RoleModel

<h1 class="h3">Role Ekle</h1>
<hr>

<div class="row">
    <div class="col-md-8">
        <form asp-controller="Admin" asp-action="RoleCreate" method="POST">
            <div asp-validation-summary="All" class="row text-danger"></div>

            <div class="form-group row mb-3">
                <label asp-for="Name" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Name">
                    <span asp-validation-for="Name" class="text-danger"></span>
                </div>
            </div>
            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2">
                    <button type="submit" class="btn btn-primary">Role Kaydet</button>
                </div>
            </div>
        </form>
    </div>
</div>
```

Formdan gelen bilgilerin işleneceği Post tipindeki RoleCreate() metodunu oluşturalım. Eğer Validationlarda bir sorun yoksa (ModelState.IsValid) ise bu durumda veritabanına Role adını (name) kaydedecek olan `_roleManager.CreateAsync` metodudur. Bu metod bizden IdentityRole sınıfından türetilmiş bir nesne isteyecektir. Bu nesneyi türetip içerisine model.Name atalım. Asenrkron bir işlem yaptığımızdan metod başlığında async ve task<> ifadelerini kullanmalıyız. Kayıt işlemimiz başarılı ise return ile buradan direk olarak RoleList() de gidebiliriz. RoleList() admin controller içinde olduğu için controller adını yazmamıza gerek kalmaz.

Eğer Role ün create işleminde bir hata oluştuysa farklı bir uygulama olsun diye oluşan hataları aşağıdaki gibi result.Errors listesinden alıp bunlarır Foreach döngüsü içinde modelState in içine ekleyip tekrar sayfaya giderken orada gösterebiliriz. Tabii bu hatalar sayfada asp-validation-summary içinde otomatik olarak gösterilecektir.

```
[HttpPost]
public async Task<IActionResult> RoleCreate(RoleModel model)
{
    if(ModelState.IsValid)
    {
        var result = await _roleManager.CreateAsync(new IdentityRole(model.Name));

        if(result.Succeeded)
        {
            return RedirectToAction("RoleList");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
        }
    }
    return View();
}
```

Şidmi RoleList sayfasını oluşturalım. Bu sayfada bir listeleme işlemi yapacağız. Dolayısı ile ürünleri listelediğimiz sayfa içinde kullandığımız tablo yapısını burada da kullanabiliriz.

RoleList.cshtml sayfasının modeli ne olacak. Bu modeli kendimiz oluşturmayalım, zaten var olan IdentityRole sınıfının yapısını kullanalım. IdentityRole sınıfının namespace ni ise _ViewImports.cshtml içine ekleyelim (`@using Microsoft.AspNetCore.Identity`). Burada geçen IEnumerable ifadesi bir liste olduğunu listenin tipininde IdentityRole olduğunu ifade etmişl oluyor.

```
@model IEnumerable<IdentityRole>
```

Sayfa içinde Role Ekleme butonunu aşağıdaki gibi oluşturabiliriz. Burada linki tanımlarken dikkat edersek action metodundaki gibi oluşturmadık. Yani oluşturulan link (`/admin/role/create`) şeklinde oldu. Oysa action metodumuzun adı (admin/rolecreate) şeklindeydi. Bu yüzden ziyaretçilere gösterilen link adresi ile onun gideceği action metodu farklı olduğundan startup içinde bu link dönüşümlerini sağlayacak Route dediğimiz kodları yazacağız.

```
<a class="btn btn-primary btn-sm" href="/admin/role/create">Role Ekle</a>
```

RoleList.cshtml sayfasına ait tüm kodları gösterirsek şu şekilde olabilir. Bazı yerleri daha sonra değiştiririz.

Views>admin>**RoleList.cshmtl**

```
@model IEnumerable<IdentityRole>

<div class="row">
    <div class="col-md-12">
        <h1 class="h3">Rolleri Listeleme</h1>
```

```
        <hr>
        <a class="btn btn-primary btn-sm" href="/admin/role/create">Role Ekle</a>

        <table class="table table-bordered mt-3">
            <thead>
                <tr>
                    <td style="width:250px">Id</td>
                    <td>Role Adı</td>
                    <td style="width:160px"></td>
                </tr>
            </thead>

            <tbody>
                @if (Model.Count() > 0)
                {
                    @foreach (var item in Model)
                    {
                        <tr>
                            <td>@item.Id</td>
                            <td>@item.Name</td>
                            <td>
                                <a href="/admin/role/@item.Id" class="btn btn-primary btn-sm
mr-2">Düzenle</a>
                                <form action="/admin/role/delete" method="post"
style="display:inline;">
                                    <input type="hidden" name="roleId" value="@item.Id" />
                                    <button type="submit" class="btn btn-danger btn-
sm">Sil</button>
                                </form>
                            </td>
                        </tr>
                    }
                }
                else
                {
                    <div class="alert alert-varning">
                        <h3>Role Yoktur</h3>
                    </div>
                }
            </tbody>
        </table>
    </div>
</div>
```

Listeleme sayfasına götürecek olan action metodunun içini dolduralım. View sayfasına giderken yanımızda içinde rollerin bulunduğu bir listeyi götürmemiz gerekir. Rollerle ilgili tüm işlemleri yürüten RoleManager sınıfının alt metodu olan _roleManager.roles bize bir liste şeklinde rolleri verecektir. View sayfasına giderken yanımızda bu listeyi götürürüz.

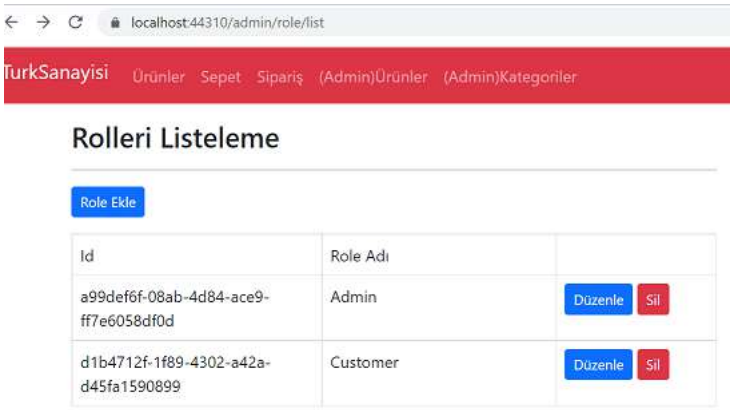
```
public IActionResult RoleList()
{
    return View(_roleManager.Roles);
}
```

Son olarak startup içindeki route desenlerini de yazalım.

```
//1---admin/role/list
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "adminroles",
        pattern: "admin/role/list",
        defaults: new { controller = "Admin", action = "RoleList" }
    );
```

```
});

//2---admin/role/list
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "adminrolecreate",
        pattern: "admin/role/create",
        defaults: new { controller = "Admin", action = "RoleCreate" }
    );
});
```

Kodlarımızı deneyelim. İlk başta liste boş gelecektir. İki tane örnek role ekleyelim.



Kullanıcıların Rollerinin Atanması

Şu ana kadar rolleri oluşturmak için alt yapıyı hazırladık. Şimdi ise hangi kullanıcı hangi role sahip olacak ve o rolle hangi linklere ulaşabilecek bu ayarlamaları yapalım.

Kullanıcıları istenen role atamak için bir RoleEdit sayfası oluşturalım. O rolle ilgili detay bilgileri geldiğinde o role sahip ve sahip olmayan kullanıcılarda gelsin. İstediğimiz kullanıcıyı o role için ekleme çıkarma yapalım.

RoleEdit sayfası için AdminController altında bir Get metodu, bir tanede Post metodu oluşturalım. Get metodundan View sayfasına giderken O role ait bilgiler ile o role sahip kullanıcıları ve diğer kullanıcıları taşıyacak olan bir model yapısına ihtiyacımız vardır. Oluşturacağımız yeni model Rolle ilgili detayları temsil edecek. Bu nedenle RoleDetails adında bir sınıf oluşturalım fakat bu sınıfı ayrı bir dosya içinde tutmayalım. Daha önceden oluşturduğumuz Models>RoleModel.cs isminde bir dosyamız vardır. Aynı dosyanın içerisine RoleDetails isminde bir sınıf daha oluşturalım. Böylece Roller ile alakalı model sınıflarımızı aynı dosya içinde tutmuş olalım. İlla her sınıf farklı dosya içinde olması gerekmemektedir.

Model içinde taşıyacağımız bilgileri üç şekilde düşünebiliriz. Hangi Role ün detaylarını göstereceksek o Role ait bilgileri taşımak için bir nesne gerekir. Bu nesne **IdentityRole** tipinde olacak. İkincisi o role ait atanmış olan kullanıcıların taşınacağı bilgi. Bu bilgiler liste şeklinde olacak (IEnumerable tipinde) ve listenin elemanları User tipinde olacak. Bunler o role ait olduğundan Members ismiyle adlandıralım. Üçüncü grup bilgi ise üye olmayan diğer tüm kullanıcılarında sayfaya taşınması gerekir. Bunları da NonMembers adı ile analım. Members olanlar checkboxlar içinde işaretli olarak diğerleri işaretsiz olarak gösterilecektir.

Models> **RoleModel.cs**

```
using Microsoft.AspNetCore.Identity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using TS.WebUI.Identity;
```

```
namespace TS.WebUI.Models
{
    public class RoleModel
    {
        [Required(ErrorMessage = "Role ismi girilmelidir")]
        public string Name { get; set; }
    }

    public  class RoleDetails
    {
        public IdentityRole Role { get; set; }
        public IEnumerable <User> Members { get; set; }
        public IEnumerable<User> NonMembers { get; set; }
    }
}
```

Şimdi bu modeli kullanarak AdminController (rolleri atama işleri adminin yapacağı işler olduğu için) içinde Get tipindeki RoleEdit() metodunda bilgilerimizi alıp View sayfasına götürelim. Metoda girişte Role in Id bilgisini almaktayız. Çünkü buraya Rolleri listelediğimiz yerden geleceğiz. Bu Id bilgisine göre role ait bilgileri asenkron metod kullanarak alacağız. Ardından members ve nonmembers için iki tane liste oluşturalım ve bu listeler içinde User bilgisisini taşıyabilsin. _userManager nesnesini kullanarak veritabanındaki tüm Userları tarayalım. Taradığımız kullanıcının verdiğim Role adındaki role ait olup olmadığını bool türünde bize verecek. Bu işlem için IsInRoleAsync() metodunu kullanacak. Burdan dönen ifade true ise members listesine, false ise nonmembers listesine eklenecek. Bu kısımda bildiğimiz basit bir if … else yapısı kullanarak yazabiliriz ama daha profesyonel bir yazım için aşağıdaki gibi bir yöntem kullanalım (boyalı gösterildi).Burada tek bir liste tanımlaması üzerinden members ve nonmembers eklemesi yapmış oluyoruz. Sistem kendi içinde otomatik olarak hangi listenin içine ekleneceğini ayarlamış oluyor. Artık oluşturduğumuz model bilgisi içerisine role bilgileri, members ve nonmembers listelerini ekleyelim.

```
    public async Task<IActionResult> RoleEdit(string Id)
    {
        var role =await _roleManager.FindByIdAsync(Id);

        var members = new List<User>();
        var nonmembers = new List<User>();

        foreach(var user in _userManager.Users)
        {
            var list = await _userManager.IsInRoleAsync(user, role.Name)
                                          ?members: nonmembers;
            list.Add(user);
        }

        var model = new RoleDetails() {
            Role = role,
            Members =members,
            NonMembers = nonmembers
        };
        return View(model);
    }
```

Metoda bağlı olan View sayfamızı oluşturalım. Sayfa içinde 12 lik colon içinde iki tane grup oluşturalım. Role ait olan kullanıcılar ve role ait olmayan kullanıcılar şeklinde olsun. Sayfa yapısı şu şekilde oldu. Detaylarını inceleyin. Burada chekboxlardan çoklu seçim yapacağız. Seçilen bu chekboxlar post edilirken liste şeklinde götülecektir. Form sayfamızdan Post metoduna giderken yanımızda hangi role üzerinde işlem yaptığımızın bilgilerini götürmemiz gerekir. Bu bilgiyi formdan taşıyabilmek için Role Id bilgisini hidden alanda tutup yanımızda götürmeliyiz. Yanımızda götüreceğimiz diğer bilgiler Eklenecek olan user ait Id bilgileri ve silinecek olan Id bilgileri olacaktır. Bunları dizi şeklinde yanımızda götüreceğiz. Ekleyecek olduğumuz chekboxları `name="IdsToAdd"` olarak,

silecek olduklarımızı da name="IdsToDelete" olarak tanımlayalım. Peki bu bilgiler Post metoduna nasıl gidecek. Taşınacak olan bu bilgileri ayrı bir model sınıfı oluşturarak taşıyabiliriz. Bu sınıfı yine RoleModel.cs dosyası içinde oluşturalım. Bu dosya içinde üç tane sınıf olmuş olacak. Oluşturduğumuz sınıfa dikkat edersek IdsToAdd ve IdsToDelete özellikleri dizi şelkinde tanımlandı. Çünkü seçtiğimiz yada çıkardığımız checkboxların Id lerini bu isimdeki değişkenin içine dizi olarak atmış olacağız. Chekbox lar üzerinde yazılı olan name="" isimli alanlar atılacak olan değişkeni, değişken içine atılan bilgi olarak da value="" değerleri kullanılmış oluyor. Yani name içindeki bilgiler modele yüklenir ve modelle post metoduna taşınmış olur.

```
public class RoleEditModel
{
    public string RoleId { get; set; }
    public string RoleName { get; set; }
    public string[] IdsToAdd { get; set; }
    public string[] IdsToDelete { get; set; }
}
```

Bu bilgileri Post metoduna taşıyacak olan View sayfamızda şu şekilde olur.

Views>Admin>**RoleEdit.cshtml**

```
@model RoleDetails

<h1 class="h3">Edit Role</h1>
<hr>
<div class="row">
    <div class="col-md-12">

        <form asp-controller="Admin" asp-action="RoleEdit">

            <input type="hidden" name="RoleId" value="@Model.Role.Id">
            <input type="hidden" name="RoleName" value="@Model.Role.Name">

            <h6 class="bg-info text-white p-1"> @Model.Role.Name : Rolüne Ekle</h6>

            <table class="table table-bordered table-sm">
                @if (Model.NonMembers.Count() == 0)
                {
                    <tr>
                        <td colspan="2">Bütün kullanılar role ait</td>
                    </tr>
                }
                else
                {
                    @foreach (var user in Model.NonMembers)
                    {
                        <tr>
                            <td>@user.UserName</td>
                            <td style="width: 150px;">
                                <input type="checkbox" name="IdsToAdd" value="@user.Id">
                            </td>
                        </tr>
                    }
                }
            </table>

            <hr>

            <h6 class="bg-info text-white p-1">@Model.Role.Name : Rolünden Çıkar</h6>
            <table class="table table-bordered table-sm">
                @if (Model.Members.Count() == 0)
                {
                    <tr>
                        <td colspan="2">Role ait kullanıcı yok.</td>
                    </tr>
                }
                else
                {
                    @foreach (var user in Model.Members)
                    {
                        <tr>
                            <td>@user.UserName</td>
                            <td style="width: 150px;">
                                <input type="checkbox" name="IdsToDelete" value="@user.Id">
```

```
                        </td>
                    </tr>
                }
            }
        </table>

        <button type="submit" class="btn btn-primary">Değişiklikleri Kaydet</button>

    </form>
</div>
</div>
```

Şimdi view sayfasından model içinde gelecek bilgileri işleyecek olan Post action metodumuzu oluşturalım.

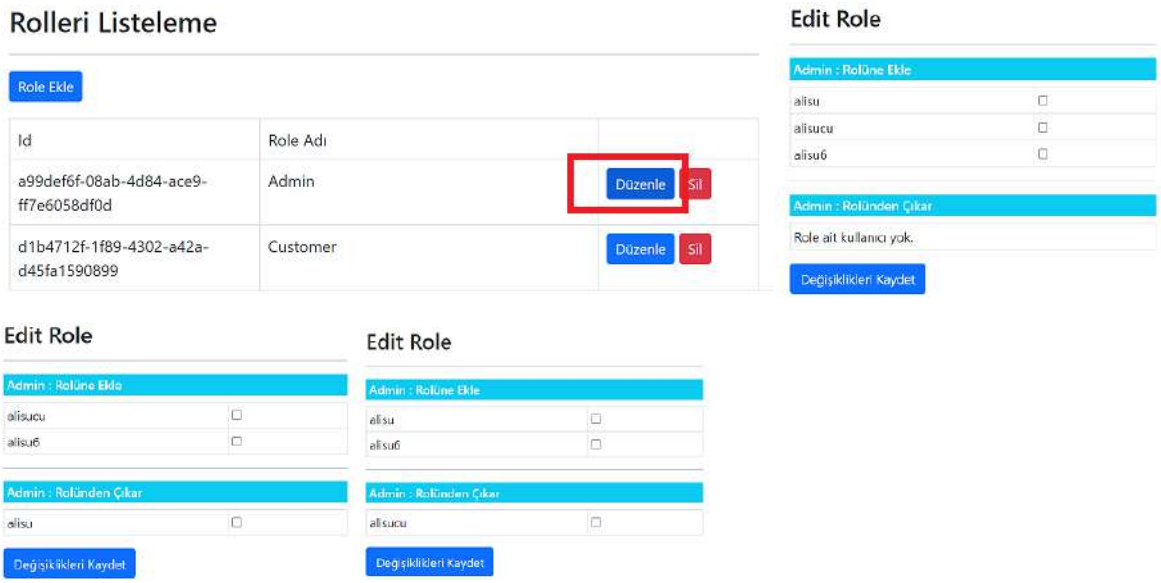
```
[HttpPost]
public async Task<IActionResult> RoleEdit(RoleEditModel model)
{
    if (ModelState.IsValid)
    {
        foreach (var userId in model.IdsToAdd ?? new string[] { })
        {
            var user = await _userManager.FindByIdAsync(userId);
            if (user != null)
            {
                var result = await _userManager.AddToRoleAsync(user, model.RoleName);
                if (!result.Succeeded)
                {
                    foreach (var error in result.Errors)
                    {
                        ModelState.AddModelError("", error.Description);
                    }
                }
            }
        }

        foreach (var userId in model.IdsToDelete ?? new string[] { })
        {
            var user = await _userManager.FindByIdAsync(userId);
            if (user != null)
            {
                var result = await _userManager.RemoveFromRoleAsync(user, model.RoleName);
                if (!result.Succeeded)
                {
                    foreach (var error in result.Errors)
                    {
                        ModelState.AddModelError("", error.Description);
                    }
                }
            }
        }
    }
    return Redirect("/admin/role/" + model.RoleId);
}
```

`foreach (var userId in model.IdsToAdd ?? new string[] { })` şeklindeki ifadenin anlamı, eğer sayfadan boş null bir dizi gelirse foreach döngüsü dönerken hata verir. Bu nedenle ?? işareltleri ile (Eğer null sa anlamında) yeni boş bir string dizi oluşturuyoruz ve bu dizinin elemanı olmadığı için foreach döngüsü hata vermez aynı zamanda dönmemiş olur.

RoleEdit sayfasına gitmek için Route desenimizi oluşturalım. Startup içine aşağıdaki şekilde yazalım.

```
//3---admin/role/{Id?}
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "adminroleedit",
        pattern: "admin/role/{Id?}",
        defaults: new { controller = "Admin", action = "RoleEdit" }
    );
});
```

Kodlarımızı deneyelim.



Rollere Sayfaların Erişim İzinlerinin Ayarlanması

Şu ana kadar Roler leri yönetmek için kod alt yapısı ve hangi role hangi kullanıcıları atayacağımızın ayarlamasını yaptık. Fakat hangi rol hangi sayfalara ulaşabilecek yada izni olmayan sayfaları erişimini nasıl engelleriz onu öğrenelim.

Burada kullanıcılara rolleri atarken kullanıcı bazlı, yada rollere kullanıcıları atarken role bazlı yapabiliriz. Biz burada rollere kullanıcıları atadık. Dolayısı ile role bazlı bir alt yapı hazırlamış olduk. Daha sonraki uygulamımızda ise kullanıcılara rolleri atayacağız.

Admincontroller sınıfının başında yazdığımı `[Authorize]` ifadesi bize bu sınıfa erişek kişilerin user giriş yapmış olmasını zorunlu kılar. Fakat hangi olduğuna bakmaz. Yani kullanıcının bilgisayarında cookie varsa erişimine izin verir. Oysa bu sınıfa Admin erişebilmeli Customer ların erişememesi gerekir. Bu kısmı aşağıdaki şekilde yazdığımızda sadece Admin rolüne ait kişiler erişebilecektir.

```
namespace TS.WebUI.Controllers
{
    [Authorize(Roles="Admin")]
    public class AdminController : Controller
    {
```

Burada Controller ın en üstüne bunu yazdığımızda içerisindeki bütün Action metodlara erişimi engeller, yani sadece Admin olanlar erişebilir. Şayet tüm controllera değilde içerisindeki farklı Action lara farklı kullanıcıların erişebilmesini istiyordak bu ifadeyi action metodların başına yazmalıyız.

Şimdi NavBar içinde (Admin) Roller şeklinde bir link oluşturalım. Bu linke tıkladığımızda admin/role/list sayfası karşımıza gelsin. Linki elle yazmak için uğraşmayalım.

```
<li class="nav-item">
    <a href="/admin/role/list" class="nav-link">(Admin)Roller</a>
</li>
```

Herhangi bir kullanıcı Admin yetkisi ile giriş yapmadıysa ve bu linke tıkladığında ne olacaktır. Bu durumda startup içinde ayarları yapılan yerdeki accessdenied yönlendirmesi çalışacaktır. Startup içindeki aşağıda adresi verilen sayfayı oluşturmamız bu gibi durumlarda oraya yönlendirme yapmamız gerekir.

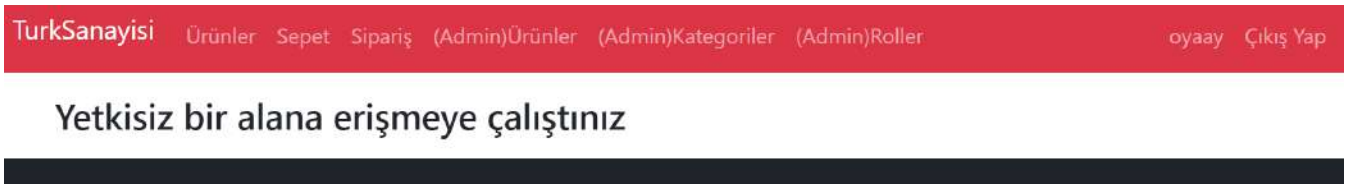
```
//Cookie ayarları
services.ConfigureApplicationCookie(option => {
    option.LoginPath = "/account/login";
    option.LogoutPath = "/account/logout";
    option.AccessDeniedPath = "/account/accessdenied";
    option.SlidingExpiration = true;
    option.ExpireTimeSpan = TimeSpan.FromMinutes(2);  //.FromDays(3);
```

O zaman account controller altında bu sayfaya gitmesi için action metod ve sayfa altyapımızı oluşturalım.

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

AccessDenied ın View sayfasını oluşturalım. Şimdilik basit bir sayfa olsun.

```
<h1 class="h3">Yetkisiz bir alana erişmeye çalıştınız</h1>
```



Accountcontroller içindeki bütün metodlara Login olan birisinin erişmesini isteyebiliriz. Dolayısı bu controllerın başına `[Authorize]`ifadesini yazmalıyız. Fakat böyle bir durumda bu controller içinde bulunan Login() metoduna erişim yapılamaz. Çünkü kişinin önce şifre girişi yapması lazım. Dolayısı ile bu Login() metoduna bütün herkes yani siteye henüz giriş yapmayan herkes ulaşabilmelidir. Dolayısı ile sadece bu action metod üzerine aşağıdaki ifadeyi yazmamız gerekir. Yani anonim olduğunu söylemeliyiz. Benzer şekilde bu controller içinde Register(), ForgotPassword() metodlarını da giriş yapmadan görebilmelidir. Ama diğer metodlar örneğin metodları ResetPassword(), Logout(), ConfirmEmail() vs giriş yapmayanların görememesi lazım. Bu controller içinde bu şekilde ayarlamaları yapmalıyız.

```
[AllowAnonymous]
public IActionResult Login()
{
```

Eğer controller içinde izin verdiklerimiz, engellediklerimizden fazla ise controller başına bir Authorize koymayız. Engellemek istediklerimizin başına [Authorize] şeklinde yazarsak en azından onlara giriş yapmadan erişimi kaldırmış oluruz. Metodların durumuna göre karar veririz. Şimdilik uygulamamızda bunları geri alalım. Accountcontroller içinde herhangi bir Authorize olmasın.

Şu ana kadar NavBar içindeki linklerin büyük bir çoğunluğunu `@if (User.Identity.IsAuthenticated)` ifadesi ile giriş yapan herhangi bir kişinin görebileceği şekilde ayarlamıştık. Oysa burada sadece admini ilgilendiren linklerde vardı. Dolayısı ile admin ilgilendiren linkleri herhangi bir giriş yapan değil sadece admin girişi yapanın görmesi gerekir. Dolayısı ile bu kısmı aşağıdaki şekilde oluşturmak daha doğru olacaktır.

```
@if (User.Identity.IsAuthenticated)
{
    <li class="nav-item">
        <a href="" class="nav-link">Sepet</a>
    </li>

    <li class="nav-item">
        <a href="" class="nav-link">Sipariş</a>
    </li>

    if (User.IsInRole("Admin"))
    {
    <li class="nav-item">
        <a href="/admin/products/" class="nav-link">(Admin)Ürünler</a>
    </li>
    <li class="nav-item">
        <a href="/admin/categories/" class="nav-link">(Admin)Kategoriler</a>
    </li>
    <li class="nav-item">
        <a href="/admin/role/list" class="nav-link">(Admin)Roller</a>
    </li>
    }
}
```

Şimdi üç tip kullanıcı için deneyelim. Ziyaretçiler (anonim), Admin, Customer olsun.

Normal ziyaretçinin gördüğü navbar.

Admin rolü ile giriş yapan alisu un gördüğü ekran

Customer rolü ile giriş yapan oyaay ın gördüğü ekran. Dolayısı ile bu kişi sepet ve sipariş bilgilerini görebilecektir. Çıkış yaptığında ise sadece ürünleri görebilecektir.

Burada sepet ve siparişleri sadece customer olanla değil tüm giriş yapanlar görebiliyor. Admin ininde sepet ve sipariş işlemi olabilir. Başka role sahip olanlarda bunu yapabilir.

Kullanıcı Listesinin Hazırlanması

Şu ana kadar Rol sayfalarından o role atayacağımız kullanıcıları belirleyebiliyoruz, fakat bunun tam tersi olarak kullanıcı sayfalarından da igili kullanıcıya istenilen rolleri atayabilmeliyiz. Yönetici olarak kullanıcının detay sayfasına gidip istenilen rolleri atayabilmemiz yada mail onayı var mı gibi bir takım bilgilerini kontrol edebilmemiz gerekir.

Bunun için bir kullanıcı listeleme sayfası oluşturalım. Buradan istenilen kullanıcının Edit butonuna tıkladığımızda o kullanıcının detay bilgilerine ulaşalım ve oradan gerekli rol atama ve ihtiyaç duyduğumuz diğer bilgilerini değiştirebilelim.

İlk olarak navbar içine bir kullanıcı listeleme linki ekleyelim. Bu linke de Admin yetkisi ulaşabilsin.

```
<li class="nav-item">
    <a href="/admin/user/list" class="nav-link">(Admin)Kullanıcılar</a>
</li>
```

Bu linkin çalıştıracağı admicontroller altında çalışacak Get tipindeki action metodumuzu oluşturalım. Metodumuz view sayfasına giderken tüm kullanıcıları götürsün. Kullanıcılar ile ilgili işlemlerde kullandığımız nesnemiz _userManager idi. Bu nesnin .Users() metodu tüm kullanıcıları veritabanından getiriyordu.

```
public IActionResult UserList()
{
    return View(_userManager.Users);
}
```

Burada userManager haricinde zaten WebUI>Identity klasörü içinde oluşturduğumuz ApplicationContext sınıfını da kullanabiliriz. Bu sınıfı aşağıdaki şekilde yazdığımızda Users yada Roles bilgilerini getirdiğini görebiliyoruz. Tabii bu contexin çalışabilmesi için Startup içinde bir servis olarak tanımlanması gerekir. O işlemi de yapmak lazım. Biz burada sadece ek bilgi olsun diye bu açıklamayı yaptık. ApplicationContext sınıfının oluşturulduğu yere bakacak olursak bu sınıfın EntityFramework ün DbContext sınıfını kullandığını görürüz.

```
public IActionResult UserList()
 {
     var context = new ApplicationContext();
     var users =context.Users;
```

`UserList.cshtml` sayfamızı oluşturmaya geçelim. Örnek sayfa olarak RoleList.cshtml sayfasını kullanalım. Sayfanın modeli olarak User ları listede tutacak (IEnumerable) bir model tanımlaması yapalım.

```
@model IEnumerable<User>
```

User ın namespace ni tüm chtml sayfalarında geçerli olması için eklediğimiz _ViewImports.cshtml içine ekleyelim.

```
@using TS.WebUI.Identity
```

View sayfamız aşağıdaki şekilde olur. `<tr class="@(item.EmailConfirmed?"":"bg-warning")">` satırı ile Emaili onaylı olmayan üyelerin arka plan rengini sarı gösterecek.

Views>Admin>**UserList.cshtml**

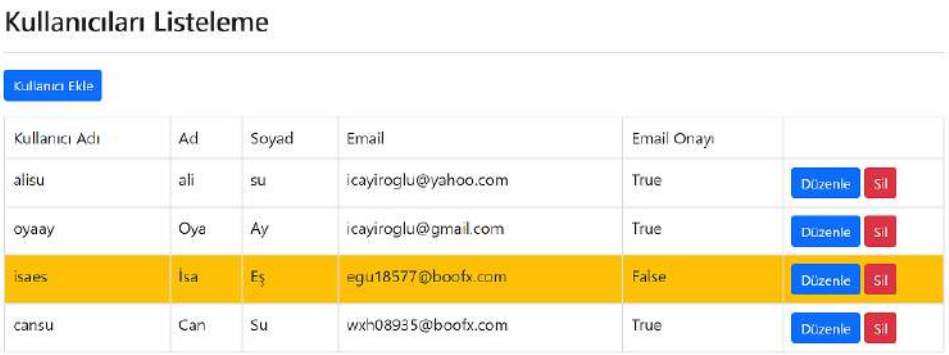
```
@model IEnumerable<User>

<div class="row">
    <div class="col-md-12">
        <h1 class="h3">Kullanıcıları Listeleme</h1>
        <hr>
        <a class="btn btn-primary btn-sm" href="/admin/user/create">Kullanıcı Ekle</a>

        <table class="table table-bordered mt-3">
            <thead>
                <tr>
                    <td>Kullanıcı Adı</td>
                    <td>Ad</td>
                    <td>Soyad</td>
                    <td>Email</td>
                    <td>Email Onayı</td>
                    <td style="width:160px"></td>
                </tr>
            </thead>

            <tbody>
```

```
                @if (Model.Count() > 0)
                {
                    @foreach (var item in Model)
                    {
                        <tr class="@(item.EmailConfirmed?"":"bg-warning")">
                            <td>@item.UserName</td>
                            <td>@item.FirstName</td>
                            <td>@item.LastName</td>
                            <td>@item.Email</td>
                            <td>@item.EmailConfirmed</td>
                            <td>
                                <a href="/admin/user/@item.Id" class="btn btn-primary btn-sm mr-2">Düzenle</a>

                                <form action="/admin/user/delete" method="post" style="display:inline;">
                                    <input type="hidden" name="userId" value="@item.Id" />
                                    <button type="submit" class="btn btn-danger btn-sm">Sil</button>
                                </form>
                            </td>
                        </tr>
                    }
                }
                else
                {
                    <div class="alert alert-varning">
                        <h3>Role Yoktur</h3>
                    </div>
                }
            </tbody>
        </table>
    </div>
</div>
```

Şimdi startup içinde Route ismini verdiğimiz link desenlerini oluşturalım. Çünkü adres satırına yazılan link deseni ile gidilecek olan Controller ve metod isimleri birebir aynı değildir. Bu yüzden desen ve adres karşılıklarını eşleştirelim.

```
endpoints.MapControllerRoute(
    name: "adminusers",
    pattern: "admin/user/list",
    defaults: new { controller = "Admin", action = "UserList" }
);

endpoints.MapControllerRoute(
    name: "adminusercreate",
    pattern: "admin/user/create",
    defaults: new { controller = "Admin", action = "UserCreate" }
);

endpoints.MapControllerRoute(
    name: "adminuseredit",
    pattern: "admin/user/{id?}",
    defaults: new { controller = "Admin", action = "UserEdit" }
);
```

Buraya kadar olan kodları denediğimizde userlist sayfamızın çalıştığını görüyoruz.



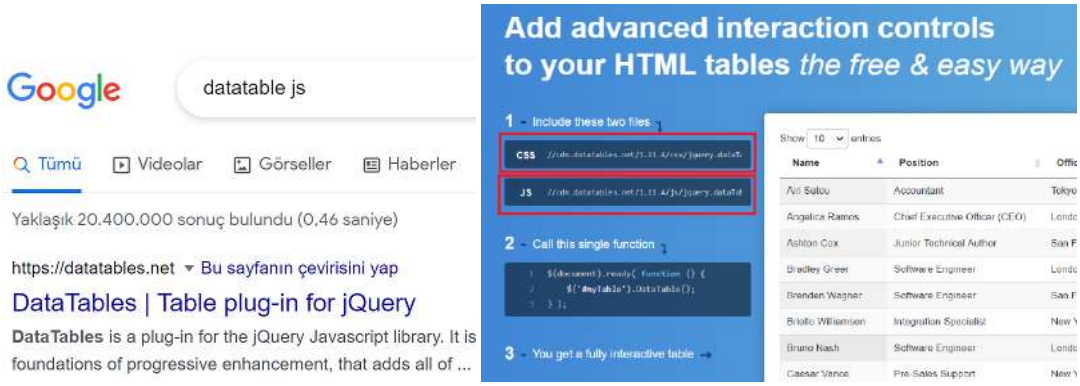
Burada mail onaysız üyeleri renkli gösterdik. İstersek telefon onayınıda aktif edebiliriz. Bu konuda altyapı Identy içerisinde vardır. Sitede çok sayıda kullanıcı olabilir. Tüm kullanıcıları tek seferde göstermek anlamsız olur. Bu

nedenle sayfalama yaptırmamız gerekir. Sayfalamanın nasıl yapıldığı Products konusunda işlenmişti. Tabii orada sayfalama yapılırken veritabanından bilgiler sayfa sayfa getiriliyordu ve böylece servera yük binmiyordu. Ayrıca ikinci bir sayfalama yöntemi ise Javascript kütüpheneleri kullanılarak istemci tarafında tarayıcı üzerinden sayfalama yapılmasıdır. Böyle bir uygulamada öncelikle tüm kullanıcılar veritabanından alınır, sadece görüntüleme yapılırken tarayıcıda sayfa sayfa gösterilir. Bu tip bir uygulamada veritabanından tüm kullanıcıları getireceğinden, kullanıcı sayısı fazla olan bir site için uygun olmayacaktır. Admin sayfaları için performans önemli değilse bu yöntem denenebilir. Fakat bu uygulamada hazır JS kütüphanesi kullanıldığından işlem kolaydır ve her sayfa geçişinde servera gidilmeyeceğinden geçişler hızlı olacaktır. Sadece tüm kullanıcıları ilk yüklemede zaman alacaktır. Biz burada iki yöntemle de sayfalamayı yapalım. İlk uygulamayı Javascript tabanl olarak deneyelim.

JavaScript Tabanlı Sayfalama

Jquery kütüphanesini kullanarak bir sayfalama uygulaması yapalım. Bu tip bir uygulama az sayıda üye varsa ve admin sayfaları için performans aranmıyorsa kullanılabilir. Ama çok sayıda üye varsa tüm üyeleri VT den çekmemek için ikinci yöntemi kullanalım.

Goole Ekranına “datatable js” yazalım ve ilk çıkan linkten Çıkan sayfadan css ve js linklerini kopyalayalım



Css linkimiz sayfadaki sayfalama yapısının görünümü belirleyecektir. Sayfanın en üstüne <style> etiketi içinde link adresimizi ekleyebiliriz fakat onun yerine _layout sayfaları için daha uygun olan Section kullanımı şeklinde yapalım. Böyle bir durumda css in sayfada konulacağı yeri _layout sayfası içindeki tanımlama belirler. style etiketleri <head> içinde olduğundan oraya koyalım. Bu durumda userlist sayfasında @section kısmının nereye yazıldığının bir önemi kalmaz.

Userlist.cshmtl sayfasında sectionın konulan yer.

```
@model IEnumerable<User>

@section Css
{
    <link rel="stylesheet" href="//cdn.datatables.net/1.11.4/css/jquery.dataTables.min.css">
}
```

_layout.cshmtl sayfasının section adresinin konulduğu yer

```
<head>
    <meta charset="UTF-8">
    <title>TurkSanayisi.com</title>
    <link href="~/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC" crossorigin="anonymous">
    @RenderSection("Css", false)
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
</head>
```

Sayfalama linkini aldığımız yerde birde Js linki vardı. Onuda benzer şekilde sayfanın içine yerleştirelim. Yalnız Js kodlarımızı sayfanın en altına koyuyorduk. Bu linkten önce (<script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>) linkinin olması gerekir. Bu link şu anda _layout içinde bulunuyor. Bu nedenle adresimizi

ise bu linkten sonra koymalıyız. Bu durumda _layout sayfası içinde section için kullandığımız referans adresimizi en altta oradaki js linkinden sonra ekleyelim. Bu linkten sonraya konulması bir zorunluluktur.

Userlist.cshtml sayfasında en üste konuldu.

```
@model IEnumerable<User>

@section Css
{
    <link rel="stylesheet" href="//cdn.datatables.net/1.11.4/css/jquery.dataTables.min.css">
}

@section Script
{
    <script src="//cdn.datatables.net/1.11.4/js/jquery.dataTables.min.js"/>

}
```

_layout.cshtml sayfasında en alta konuldu.

```
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.8.1/css/all.css">
    <script src="https://code.jquery.com/jquery-3.4.1.min.js" integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
crossorigin="anonymous"></script>
    @RenderSection("Scripts", false)
</body>
</html>
```

Henüz işimiz bitmedi. İnternetteki sayfadan Jquery scriptin olduğu yeride kopyalayı userlist sayfasındaki script bloğu içine ekleyelim.


```
@section Script
{
    <script src=
"//cdn.datatables.net/1.11.4/js/jquery.dataTables.min.js"> </script>
    <script>
        $(document).ready( function () {
        $('#myTable').DataTable();
        } );
    </script>
}
```

Burada verilen “myTable” ismi sayfamızda ulaşağımız tablonun adıdır. Dolayısı ile aşağılarda kullanacağımız o tablonun Id bilgisini myTable yapmalıyız. Böylece DataTable özelliğini bu tablo üzerinde çalıştıracak. Tekrar hatırlacak olursak userlist sayfasındaki scriptler mutlaka jquery kütüphanesinden sonra gelmelidir. O kütüphanede _layout sayfasına konulduğuna göre section adresi ondan sonra gelmelidir.

```
    <table id="myTable" class="table table-bordered mt-3">
```

İlgili web sitesinde diğer seçenekler için aşağıdaki butona tıklanarak diğer seçenekler ayarlanabilir. Örneğin sayfalama başına kaç kayıt olacağı gibi.

Kodlar denendi fakat çalışmadı. Daha öncede Js kodları çalışmamıştı. JS kullanarak Validation yaparkende çalışmamıştı. Sorun bulununca bu yapılanlar çalışacaktır.

O zaman biz daha önce yaptığımız ve olması gereken yöntemle sayfalamamızı yapalım. Yani

Asp.net tabanlı sayfalama-(userList de sayfalama)

Kullanıcıların bilgilerini sayfalama şeklinde gösterirken sonuçta her tıklanılan sayfalama butonu bir link içermekte ve bu link içerisinde sayfa numarası controllera gitmektedir. Bu link desenimiz aşağıdaki şekilde olsun.

"admin/user/list?Page=1"

şeklinde link deseni (pattern) olduğunda Admin controller altında UserList() metoduna gitsin. Tabii burada ? işaretinden sonraki kısım QueryString (sorgulama metni) olduğu için bu kısım metod girişinde parametreler ile alınacak. Dolayısı ile burada bir tane Page parametresi götürmüş olacağız. Ondan önceki route kısmı ise gidilecek metodun neresi olacağını belirleyecektir. Aslında bu şekilde yazılması sadece SEO açısından (arama motoru optimizasyon) ihtiyaç olduğundan bu desen kullanılıyor. Fakat gidilecek olan controller ve metod ise Startup içinde yazılan kod deseni ile belirlenmiş oluyor. Buna göre startup içine daha önce yazdığımız aşağıdaki desen yapımızı aynen kullanabileceğiz.

```
endpoints.MapControllerRoute(
    name: "adminusers",
    pattern: "admin/user/list",
    defaults: new { controller = "Admin", action = "UserList" }
);
```

Metod girişimizi şu şekilde yapabiliriz. Burada page parametresine 1 dememizin sebebi herhangi bir page ifadesi link deseni içinden gelmez ise ilk sayfayı göstersin diye yazıldı. Şage ?page=2 gibi parametre gelirse ilk atanan ifadeden sonra 2 onun yerine atanır. pageSize parametresinide burdan verelim. Dışarıdan bu parametre daha sonradan farklı gelirse ona göre değişecektir.

```
public IActionResult UserList(int page = 1, int pageSize = 3)
```

UserList() metodumuzdan View sayfasına bilgileri götürürken Users sınıfını model olarak götüreceğiz. View sayfasında görüntülemeyi ise Asp'nin hazır bir nesnesi olan "PagedList" sınıfını kullanacağız

Bu sınıfı projemizin içine kurmamız gerekir. Tools>Nuget Package Maneger>Package Manager Consule kullanarak (https://www.nuget.org/packages/PagedList.Core.Mvc/) adresinden alınan linki çalıştırıp kuralım.

Install-Package PagedList.Core.Mvc -Version 3.0.0

AdminController içine yazdığımız listeleme sayfasına götürecek olan Get metodunun içeriği aşağıdaki şekildedir.

```
public IActionResult UserList(int page = 1, int pageSize = 3)
{
    var users = _userManager.Users;

    PagedList<User> model = new PagedList<User>(users, page, pageSize);

    return View(model);
}
```

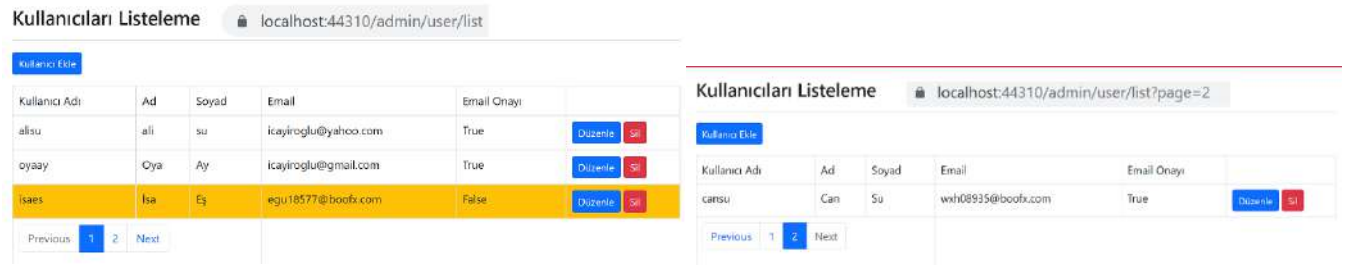
UserList.cshtml view sayfasının içeriği ise

```
@model PagedList.Core.IPagedList<User>

**Ara kısımlar çıkarıldı. Kodları Bir önceki konuda vardır.**
                <tr>
                    <td colspan="3" align="center">
                        <pager list="@Model" asp-controller="Admin" asp-action="UserList" />
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

View sayfalarının NameSpacelerini tutan _ViewImports.cshtml dosyasının içine eklenen satırlar.

```
@addTagHelper *,PagedList.Core.Mvc

@using PagedList
```

Kodları deneyelim.



Kullanıcı (User) Bilgilerinin Güncellenmesi

Kullanıcı bilgilerini listeledikten sonra liste üzerinde Düzenle butonuna tıkladığımızda kullanıcının bilgilerini düzenleyeceğimiz sayfa gidip oradan düzenleme yapabilmeliyiz. Düzenle butonuna tıkladığımızda linkimizi (href="/admin/user/@item.Id") şeklinde oluşturduk. Bu linke tıkladımızda gideceğimiz yer Admincontroller altında UserEdit() metodu olacaktır. Bu metoda gidebilmesi için Startup.cs içinde adres desenimiz şu şekilde olur. Bunu daha önce oluşturmuştuk.

```
endpoints.MapControllerRoute(
    name: "adminuseredit",
    pattern: "admin/user/{id?}",
    defaults: new { controller = "Admin", action = "UserEdit" }
);
```

Şimdi bu UserEdit metodu oluşturalım. Bu metod asenkron tipte olacaktır. Dışarıdan kullanıcının Id bilgisini alacağız ve bu Id ye göre sorgulama yapacağız ve kullanıcının bilgilerini Edit sayfasına taşıyacağız fakat sadece kullanıcı bilgileri olsaydı zaten User nesnesi üzerinden bilgileri taşıyabilirdik. Fakat aynı zamanda kullanıcının bağlı olduğu rol bilgilerini ve bağlı olmadığı tüm role bilgilerine de ihtiyacımız vardır. Bu bilgileri de yanımızada götürmemiz gerekiyor. Dolayısı hem role bilgileri hemde user bilgilerini bir paket haline getirip taşımak için yeni bir model sınıfı oluşturmalıyız. Şimdi bu model sınıfını oluşturalım. Model klasörü altında UserDetailsModel isminde bir model sınıfı oluşturalım. Sınıf içerisinde kullanıcı için belirlediğimiz temel alanların yanında seçilen role bilgilerini bir liste şeklinde (IEnumerable) tutalım. Model sınıfımız şu şekilde olacaktır.

Models> **UserDetailsModel.cs**

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

namespace TS.WebUI.Models
{
    public class UserDetailsModel
    {
        public string UserId { get; set; }
        public string UserName { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
```

```
        public string EmailConfirmed { get; set; }
        public IEnumerable<String> SelectedRoles { get; set; }

    }
}
```

Bu bilgileri Get tipindeki UserEdit() metodu içinde Id bilgisine dayanarak alıp View sayfasına götürelim.

```
public async Task<IActionResult> UserEdit(string id).
```

Dışarıdan alınan id yi kullanarak user bilgilerini asenkron bir metod kullanarak alalım.

```
var user = await _userManager.FindByIdAsync(id);
```

Eğer user bilgisi null a eşit değilse bu durumda user ın role bilgilerini alabiliriz. Ardından veritabanındaki tüm roles bilgilerini de alalım. Yalnız sadece role adlarını (Name) string olarak alıp liste içinde tutalım.

```
var selectedRoles = await _userManager.GetRolesAsync(user);
var roles = _roleManager.Roles.Select(i => i.Name);
```

User bilgilerini sayfaya model içinde taşıyalım. İçerisinde kullanıcının kendi seçtiği role bilgileri de vardı. Fakat ayrıca sayfaya tüm role bilgilerini de taşımamız gerekiyor. Bunlarıda ViewBag içinde taşıyalım. ViewBag kullanılırken nokta koyup hangi değişken içinde taşıyacaksak onun adı yazılır. ViewBag.roles şeklinde olabilir.

Get tipindeki UserList() metodumuzun son hali şu şekilde oldu.

```
        public async Task<IActionResult> UserEdit(string id)
        {
            var user = await _userManager.FindByIdAsync(id);

            if(user!= null)
            {
                var selectedRoles = await _userManager.GetRolesAsync(user);
                var roles = _roleManager.Roles.Select(i => i.Name);

                ViewBag.roles = roles;

                return View(new UserDetailsModel()
                {
                    UserId = user.Id,
                    UserName = user.UserName,
                    FirstName = user.FirstName,
                    LastName = user.LastName,
                    Email = user.Email,
                    EmailConfirmed = user.EmailConfirmed,
                    SelectedRoles = selectedRoles
                }) ;
            }
            return RedirectToAction("~/admin/user/list");
        }
```

UserEdit sayfamızı oluşturalım. Bunun için ProductEdit sayfasından örnek format alabiliriz. Gerekli düzenlemeleri yaparken <form >etiketi içindeki (enctype="multipart/form-data") ifadesini kaldırdık. Bu ifadeyi bir dosya gönderme işlemi yaptığımızda yazıyorduk. Hidden şeklinde tanımlanan UserId kısmı post ederken action da yine ihtiyaç olacağı için kullanıcıya göstermeden form içinde tutmak için hidden olarak sakladık (<input type="hidden" name="UserId" value="@Model.UserId">).

(@foreach (var rolename in ViewBag.Roles)) döngüsü ile tüm seçilen/seçilmeyen role bilgilerini ViewBag içinden sırayla okuyup rolename değişkeni içine atıyoruz. ViewBag içinde tutulan bilgiler sadece veritabanındaki Role adları idi. Döngü içinde tüm role adlarını göserirken kullanıcının atanmış olduğu role adlarının hangisi olduğunu gösermek için chekbox içinde (@(Model.SelectedRoles.Any(i => i == rolename) ? "checked" : "")>) satırı kullanıldı.

Tüm rolleri alt alta sıraladıktan sonra (kullanıcının atanmış oldukları seçili gösterildi) araya bir çizgi çizilip altında kullanıcının Email Confirmed özelliğini admin tarafından değiştirilebilmesi için (<input asp-for="EmailConfirmed" type="checkbox" class="custom-control-input" />) satırı eklendi.

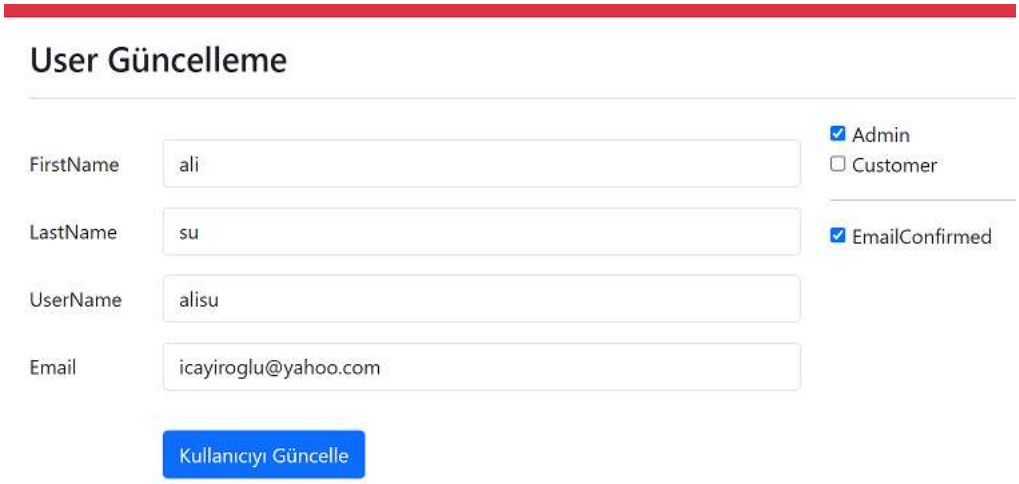
UserEdit.cshtml sayfamış aşağıdaki şekilde oldu.

Views>Admin>**UserEdit.cshmtl**

```
@model UserDetailsModel

<h1 class="h3">User Güncelleme</h1>
<hr>

<form asp-controller="Admin" asp-action="UserEdit" method="POST" enctype="multipart/form-data">

    <div class="row">
        <div class="col-md-8">
            <div asp-validation-summary="All" class="row text-danger"></div>

            <input type="hidden" name="UserId" value="@Model.UserId">

            <div class="form-group row mb-3">
                <label asp-for="FirstName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="FirstName">
                    <span asp-validation-for="FirstName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="LastName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="LastName">
                    <span asp-validation-for="LastName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="UserName" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="UserName">
                    <span asp-validation-for="UserName" class="text-danger"></span>
                </div>
            </div>

            <div class="form-group row mb-3">
                <label asp-for="Email" class="col-sm-2 col-form-label"></label>
                <div class="col-sm-10">
                    <input class="form-control" asp-for="Email">
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
            </div>


            <div class="form-group row mb-3">
                <div class="col-sm-10 offset-sm-2 mt-3">
                    <button type="submit" class="btn btn-primary">Kullanıcıyı Güncelle</button>
                </div>
            </div>

        </div>

        <div class="col-md-4">

            @foreach (var rolename in ViewBag.Roles)
            {
                <div class="custom-control custom-checkbox">
                    <input type="checkbox"
```

```
                        class="custom-control-input"
                        name="selectedRoles"
                        value="@rolename"
                        id="@rolename"
                        @(Model.SelectedRoles.Any(i => i == rolename) ? "checked" : "")>
                    <label class="custom-control-label" for="@rolename">@rolename</label>
                </div>
            }
            <hr />

            <div class="custom-control custom-checkbox">
                <input asp-for="EmailConfirmed" type="checkbox" class="custom-control-input" />
                <label asp-for="EmailConfirmed" class="custom-control-label"></label>
            </div>
        </div>
    </div>
</form>
```

Programı çalıştırdığımızda güncelleme formu karımıza gelmektedir. Burada yapacağımız bir değişliklerden sonra butana bastığımızda bilgilerin Post tipindeki Action metoduna götürülmesi gerekir.



Şimdi Post metodumuzu oluşturalım. İlk olarak sayfamızın modelini alarak asenktron tipindeki post metodumuzu oluşturuyoruz.

```
[HttpPost]
public async Task<IActionResult> UserEdit(UserDetailsModel model)
```

Form içinde seçili olan role chekboxlarının değerleri (value) `name="selectedRoles"` içinde bir string dizi şeklinde tutuluyordu. Bunlar sayfanın modeli içine konuluyor. Model içinde yerleri vardı (`public IEnumerable<String> SelectedRoles { get; set; }`). Dolayısı ile Post metoduna model içinde bu bilgiler gelirken, metod girişinde string şeklinde bir diziyi beklediğimiz söylememiz gerekir. Yani giriş paramatrelerini aşağıdaki gibi yazmalıyız.

```
public async Task<IActionResult> UserEdit(UserDetailsModel model, string [] SelectedRoles)
```

Ardından ilk olarak formdan gelen bilgilerin Validation ları doğrumu kontrol edelim. Eğer model valid ise kullanıcı bilgilerini veritabanından alalım. Model içinde kullanıcının UserId bilgisi bulunmakta. Bu bilgiyi kullanalım.

```
if(ModelState.IsValid)
{
    var user = await _userManager.FindByIdAsync(model.UserId);
```

model üzerinden gelen bilgiler user içine atılarak yeni haliyle veritabanında güncelleme işlemini yapıyoruz. Bu işlemleri yaparken _userManager sınıfını kullanıyoruz. İstersek veritabanı işlemlerini Identiy altındaki Context üzerindende yapabilirdik. Buradaki context yapısı daha önce Data katmanında product ve category için kullandığımız Context sınıfına benzemektedir.

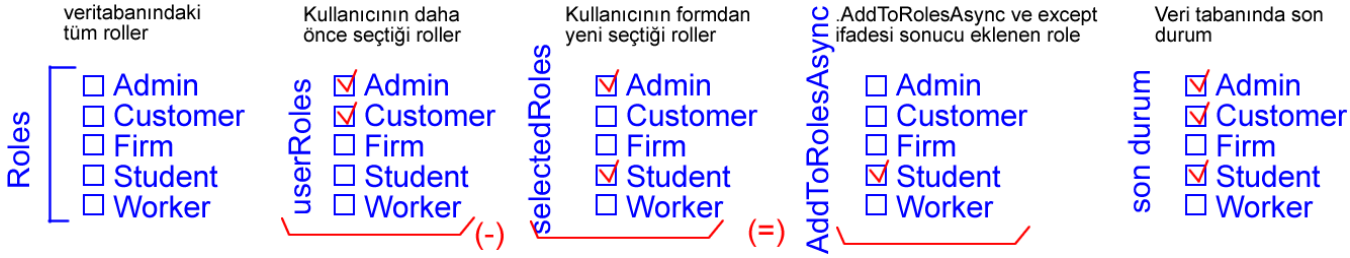
```
if(user !=null)
```

```
    {
        user.FirstName = model.FirstName;
        user.LastName = model.LastName;
        user.UserName = model.UserName;
        user.Email = model.Email;
        user.EmailConfirmed = model.EmailConfirmed;

        var result = await _userManager.UpdateAsync(user);
```

User içerisine yeni bilgileri atıp güncellemeyi yaptık. Eğer sonuç başarılı ise kullanıcının role bilgilerini kaydedelim. Başarısızca bir hata mesajıda yayınlayabiliriz. Kullanıcının role işlemlerini yapıyorsak _userManager üzerinden roles bilgilerini değiştiriyoruz. (`var userRoles = await _userManager.GetRolesAsync(user);`) satırı ile kullanıcının daha önceden ayarlanmış role bilgileri gelecektir. Eğer veritabanından tüm roller üzerinde işlem yapıyorsak _roleManager kullanmamız gerekir.

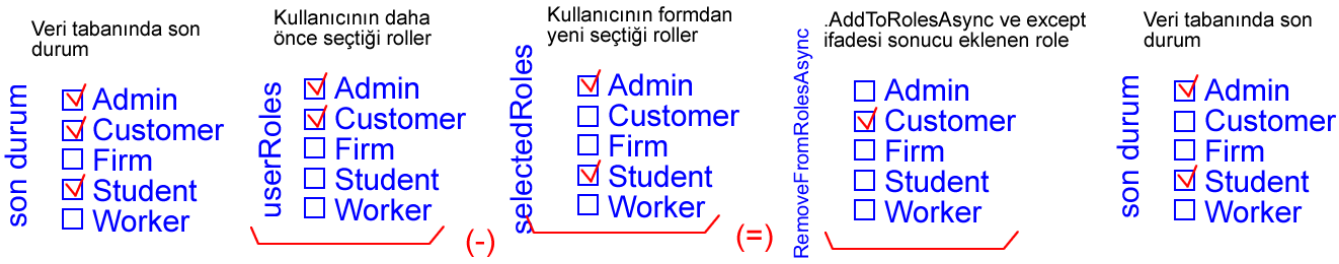
(`selectedRoles = selectedRoles ?? new string[] { };`) satırında eğer dışarıdan aldığımız seçilen rollerin bulunduğu dizi eğer null ise içerisine boş bir dizi atıyoruz ki, null hatası almayalım. Burada geçen ?? işareti "eğer null ise" anlamındadır.

(`selectedRoles`) ifadesi içinde kullanıcının form üzerinden seçtiği roller bulunmaktadır. Fakat veritabanında da daha önceden seçtiği rol kayıtları vardır. Formdan gelen rollerin bazıları veritabanında daha önceden kişi için kayıtlıdır. Bu sonradan eklenen rolleri ise `AddToRolesAsync()` ifadesi ile veritabanına aktaracağız. Bu ifade bize hangi user için yapacağını soracaktır. İçerisine user bilgimizi veriyoruz. Ayrıca yanında formdan seçtiğimiz rolleride veriyoruz. İfadeyi `(user, selectedRoles)` bu şekilde yazarsak veritabanına yeni eklenecek rolleri ekler. Fakat kullanıcının veritabanında kayıtlı daha önceden olan rolleri de vardır. Bu rolleri çıkarıp sadece yeni seçtiklerini eklememiz gerekir. Dolayısıyla daha önceden seçili olanları bu yeni listeden çıkarırsak, yeni ilk defa seçilmiş olanlar elimizde kalır. İşte formdan gelen listeden, daha önceden seçili olanları çıkarmak içinde except ifadesini kullanıyoruz. Bu ifade bizden hangi listeyi çıkaracağını sorar. Bu çıkaracağımız liste ise selectedRoles ifadesidir. Ardından oluşan son durum sadece sonradan eklenecek olan yeni listedir. Bu listeyi de diziye çeviriyoruz ve tipini de string ayarlıyoruz. Bunu bir örnekleyerek gösterelim.




```
await _userManager.AddToRolesAsync(user, selectedRoles.Except(userRoles).ToArray<string>());
```

Burada yapılan uygulamada veritabanına formdan ilk defa seçilen role eklendi (student). Peki kullanıcı daha önceden kayıtlı rollerinden bir tanesinde çıkarmış (Customer) fakat son durumda bu role hala veritabanında kaldı. Bununda çıkarılması gerekir. Aşağıdaki satırda o işlemi yapacaktır. Burada ise except ifadesi ile neleri hariç tutacağımızı vereceğiz.



```
await_userManager.RemoveFromRolesAsync(user, userRoles.Except(selectedRoles).ToArray<string>());
```

İşlem bittikten sonra kullanıcıları listeleme sayfasına gönderelim. Bunun RedirectToAction değil link adresini vererek yapalım. Yani "Redirect" metodunu kullanalım.

```
return Redirect("/admin/user/list");
```

Bu kısımda bir mesaj verebiliriz yada bir hata kontrolü gibi işlemlerde yapabiliriz.

Metodumuzun son hali aşağıdaki şekilde oldu.

```
    [HttpPost]
    public async Task<IActionResult> UserEdit(UserDetailsModel model, string [] selectedRoles)
    {
        if(ModelState.IsValid)
        {
            var user = await _userManager.FindByIdAsync(model.UserId);

            if(user !=null)
            {
                user.FirstName = model.FirstName;
                user.LastName = model.LastName;
                user.UserName = model.UserName;
                user.Email = model.Email;
                user.EmailConfirmed = model.EmailConfirmed;

                var result = await _userManager.UpdateAsync(user);

                if(result.Succeeded)
                {
                    var userRoles = await _userManager.GetRolesAsync(user);
                    selectedRoles = selectedRoles ?? new string[] { };
                    await _userManager.AddToRolesAsync(user,
selectedRoles.Except(userRoles).ToArray<string>());
                    await _userManager.RemoveFromRolesAsync(user,
userRoles.Except(selectedRoles).ToArray<string>());
                    return Redirect("/admin/user/list");
                }
            }
            return Redirect("/admin/user/list");
        }
        return View(model);
    }
```

Kodları denediğimizde çalıştığını göreceğiz.

Admin Kullanıcısının Eklenmesi

Sitemizi yayına aldığımız bu veritabanı olmayacak. Dolayısı ile sitenin yönetimi için daha baştan bir Admin rolüne ihtiyaç olacaktır. Bu işlem için Identity içinde bir seed (çekirdek) yapı oluşturmamız gerekiyor. Öncelikle database mizi silelim. Yani database miz hiç olmamayacak. Bu işlem için ister Data katmanı içinde ister WebUI katmanı içinde aşağıdaki komutumuzu çalıştıralım. Öncesinde database kapatılmalıdır.

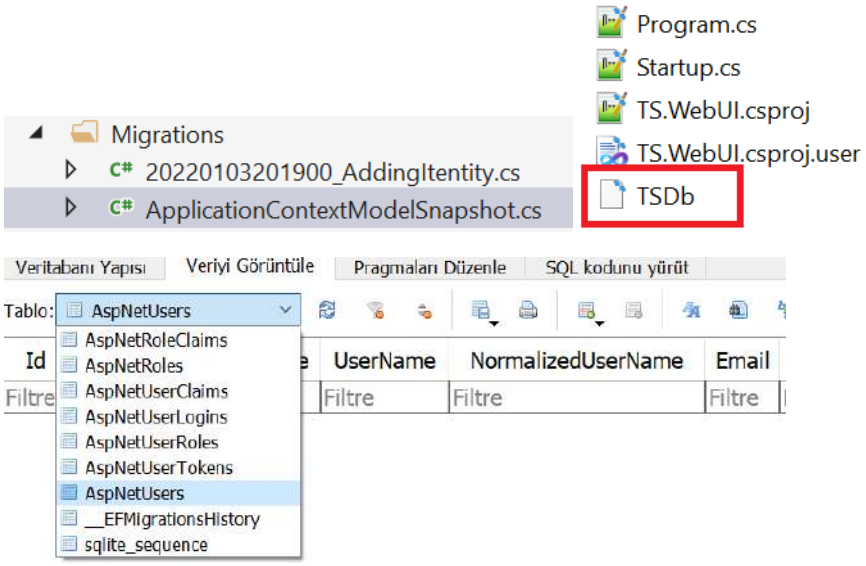
dotnet ef database drop --force

```
C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI>dotnet ef database drop --force
Build started...
Build succeeded.
Dropping database 'main' on server 'TSDb'.
Successfully dropped database 'main'.
```

Artık dabase miz yoktur. Klasöre baktığımızda göremeyiz. Ardından aşağıdaki komutu çalıştırırsak Identity ile ilgili daha önceden oluşturduğumuz migrationlar çalışır. Aynı zamanda veritabanımızın oluştuğunu, İçerisini açtığımızda ise sadece kullanıcı tablolarının (identity tablolarının)oluştuğunu ve içlerinin boş olduğunu görürüz.

dotnet ef database update

```
C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI>dotnet ef database update
Build started...
Build succeeded.
Done.
```



Burada veritabanı içinde product ve category tablolarınında olması gerekir. Bu tabloları oluşturmak için data katmanına (TS.Data) geçmeliyiz. Burada database yine update edersek çok önceden oluşturduğumuz Migrationlar çalışacaktır ve veritabanı içinde tablolarımız oluşacaktır. Update komutunu data katmanınında kullanırken, bu data katmanını yönetecek olan startup projeyide belirtmeliyiz. Komutun kullanımı aşağıdaki şekilde olur.

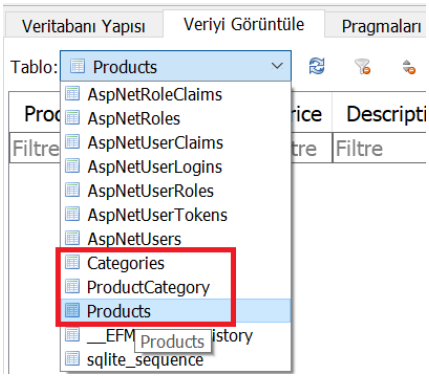
dotnet ef database update --startup-project ../TS.webUI (**Bu hatalı aşağıdakini deneyin**)

Fakat Katmanlarımız içerisinde birden fazla context varsa Solutionımız hangi context üzerinden database bağlanacağını bilemez. Solution içerisinde 2 tane context miz var. Birincisi "TS.data>Concrete>EfCore>Tscontext.cs" ikincisi ise "TS.WebUI>Identity>AplicationContext.cs" dir. İşlem bu iki context i de göreceğinden hata alırız. O yüzden veritabanına hangi context üzerinden bağlanacağımızı da belirtmeliyiz.

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

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.webUI --context TSContext
Build started...
Build succeeded.
Applying migration '20211119212740_InitialCreate'.
Applying migration '20211204213231_AdColumnProductIsHome'.
Done.
```

Migrationlar çalıştı. Şimdi database içinde tablolarımız oluşmuş mu görelim. Üç tane tablomuzu bu migrationlar ile oluşturulduğunu görüyoruz.



Şu anda veritabanı tablolarının için hep boştur. Site ilk çalıştığında bir admin kullanıcısı olmadığından siteye hiçbir şekilde giriş yapamayız. Dolayısı ile en azından başlangıç verilerini daha olsa girmek için bir Admin kullanıcısına ihtiyaç vardır.

Bu konuda startup dosyanın içerisinde, program geliştirme aşamasında kullanmak üzere bir SeedDatabase isminde içerisinde veritabanında deneme amaçlı verilerin olduğu bilgileri yüklüyorduk. Yani veritabanına denemek için bazı bilgileri giriyorduk. Ama bu bilgiler deneme amaçlı konuluyordu ve bu yüzden bloğun başında IsDevelopment ifadesi true iken çalışıyordu. Oysa biz gerçek yayına geçtiğimizde artık bu blok yapısını kullanmayacağız.

```
//Kodlar Geliştirme aşamasında çalıştırılırken. Yayınlanmaya daha geçilmediyse
    if (env.IsDevelopment())
    {
        SeedDatabase.Seed();

        app.UseDeveloperExceptionPage();
    }
```

Kullanacağımız sabit ilk bilgileri WebUI>AppSettings.Json dosyası içine ekliyoruz. Bu dosya içerisine daha önce de mail bilgilerini eklemiştik. Benzer şekilde bir sekme daha oluşturup içerisine başlangıç admin bilgilerini ekleyelim. Burada birden fazla başlangıç user olabilr. O nedenle bunlarıda bir üst parantez olan “Data” içine alıyoruz.

```
"Data":
  {
    "AdminUser": {
      "username": "admin",
      "firstname": "Ali",
      "lastname": "Su",
      "email": "icayiroglu@yahoo.com",

      "password": "Dene%123",
      "role": "admin"
    }
  },
```

Başlangıç admin verileri için WebUI>Identity klasörü içinde SeedIdentity.cs isminde bir class dosyası ve yapısı oluşturalım. Bu class yapımız static olacak. İçerisinde aşağıdaki gibi bir metod oluşturalım. Bu metod dışarıdan user bilgilerini ve role bilgilerini alsın. Ayrıca biraz sonra AppSettings içine yazacağımız Admin bilgilerini buraya

getirmek için kullanacağımız IConfiguration nesnesini yazıyoruz. Configuration üzerinden AppSettings.json dosyasına ulaşacağız ve oradan admin için verilen başlangıç bilgilerini alacağız.

(`if (await userManager.FindByNameAsync(username) == null)`) ifadesiyle içeri aldığımız kullanıcı var mı kontrol edelim. Yoksa yani null ise o zaman bu kullanıcıyı veritabanına ekleyebiliriz.

Kullanıcıyı oluşturmadan önce bir role oluşturmamız gerekiyor.

```
await roleManager.CreateAsync(new IdentityRole(role));
```

Devamında tüm kodlarımız şu şekilde olacaktır.

WebUI>Identity>**SeedIdentity.cs**

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using System.Threading.Tasks;

namespace TS.WebUI.Identity
{
    public static class SeedIdentity
    {
        public static async Task Seed(
            UserManager<User> userManager,
            RoleManager<IdentityRole> roleManager,
            IConfiguration configuration)
        {
            var username = configuration["Data:AdminUser:username"];
            var email = configuration["Data:AdminUser:email"];
            var password = configuration["Data:AdminUser:password"];
            var role = configuration["Data:AdminUser:role"];

            if (await userManager.FindByNameAsync(username) == null)
            {
                await roleManager.CreateAsync(new IdentityRole(role));

                var user = new User()
                {
                    UserName = username,
                    FirstName = "Ali",
                    LastName = "Su",
                    Email = email,
                    EmailConfirmed = true

                };

                var result = await userManager.CreateAsync(user, password);

                if (result.Succeeded) //eğer kullanıcı oluşturulduysa, rollerini ata.
                {
                    await userManager.AddToRoleAsync(user, role);
                }
            }
        }
    }
}
```

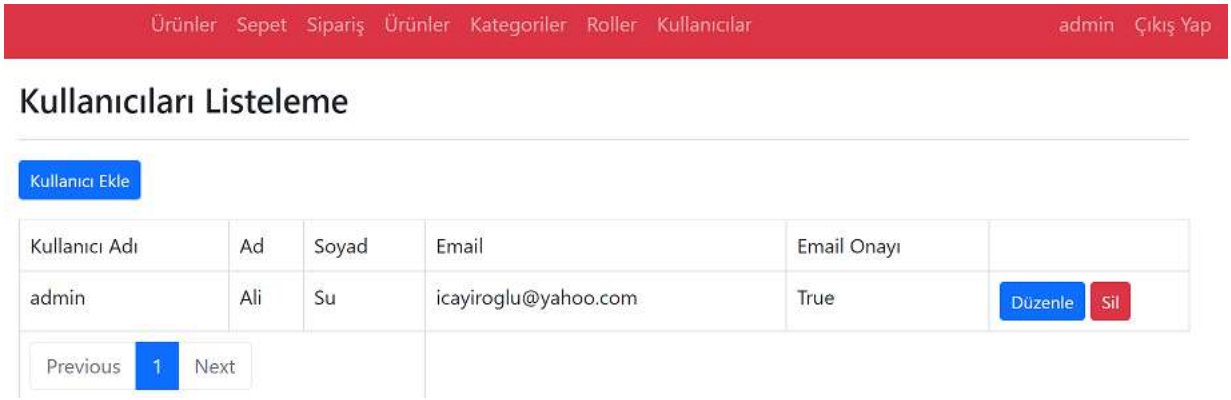
Oluşturduğumuz bu class yapısını Startup içindeki Configure metodu içinde çalıştırmamız gerekiyor. Fakat oraya SeedIdentity.Seed() yazdığımızda bizden usermanager, rolemanager ve configuration bilgilerini isteyecektir. Bu bilgiler ise içinde bulunduğu Configure() metoduna giriş kısmında verilmelidir.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, UserManager<User>
userManager, RoleManager<IdentityRole> roleManager, IConfiguration configuration)
```

Dıştaki configure() metotdan aldığımız bu bilgileri seed() metoduna vereceğiz. Bu metod asenkron olduğu için sonuna .wait() özelliğini ekleyeceğiz. Aşağıdaki kodları Configure içinde Rout desenlerini verdiğimiz kısımdan sonrasına yazmalıyız.

```
SeedIdentity.Seed(userManager, roleManager, configuration).Wait();
```

Kodlarımızı çalıştıralım. Sitemizin çalıştığını görüyoruz. Kullanıcılara baktığımızda hiç kayıt yapmadan admin kullanıcısının eklendiğini görüyoruz. Tabiki sayfalara giriş yapabilmek içinde Admin kullanıcısı da aktif çalışıyordu.



Bundan sonra artık Cart (Alışveriş sepeti) ve Order (Sipariş verme) uygulaması aşamasına geçelim. Bu kısım ayrı bir bölüm halinde anlatılmış olsun.

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