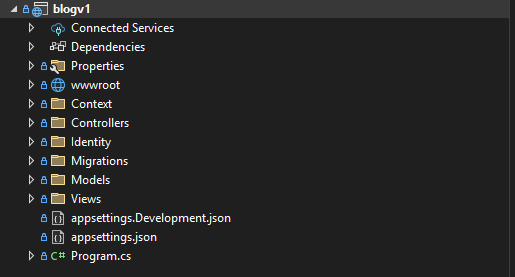
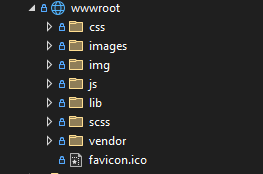
**MVC Blog Project Documentation**

**Muhammet Emir Yüce**

**05/09/2025**



Wwwroot daha frontend tarafı



# Context



* **using blogv1.Models;**

Blogv1 projemızın altındaki listeler model

**model = veritabanı tablosunu temsil eden bir sınıf** diyebiliriz. Bu sınıfı kullanarak, veritabanındaki her bir blog yazısı için bir **nesne (object)** oluşturup, bu nesneler üzerinden kolayca işlem yaparsın.

* **public class BlogDbContext : DbContext**

public class BlogDbContext : DbContext: Bu satır, BlogDbContext adında bir sınıf tanımlar ve onu EF Core'un ana sınıfı olan **DbContext**'ten türetir. Bir sınıfın başka bir sınıftan türemesi, DbContext'in tüm özelliklerini ve yöntemlerini miras almasını sağlar.

* **protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder): Bu yöntem, veritabanı bağlantı ayarlarının yapıldığı yerdir.

override anahtar kelimesi, bu yöntemin DbContext sınıfından miras alınan bir yöntemi geçersiz kıldığını (yeniden tanımladığını) belirtir.

optionsBuilder.UseSqlServer(...): Bu satır, uygulamanızın **SQL Server** veritabanına bağlanacağını belirtir. Parantez içindeki metin, veritabanı bağlantı dizesidir (connection string).

**"Data Source =NB3\SQLEXPRESS"**: Veritabanı sunucusunun adını belirtir.

**"database=blogV1"**: Bağlanılacak veritabanının adını belirtir.

**"Integrated Security=True"**: Uygulamanın Windows kimlik doğrulamasını kullanarak veritabanına bağlanacağını belirtir, yani kullanıcı adı ve şifreye gerek yoktur.

**"TrustServerCertificate=True"**: Sunucu sertifikasına güvenileceğini belirtir, bu genellikle yerel geliştirme ortamlarında kullanılır.

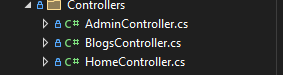
* **public DbSet<Blog> Blogs { get; set; }**

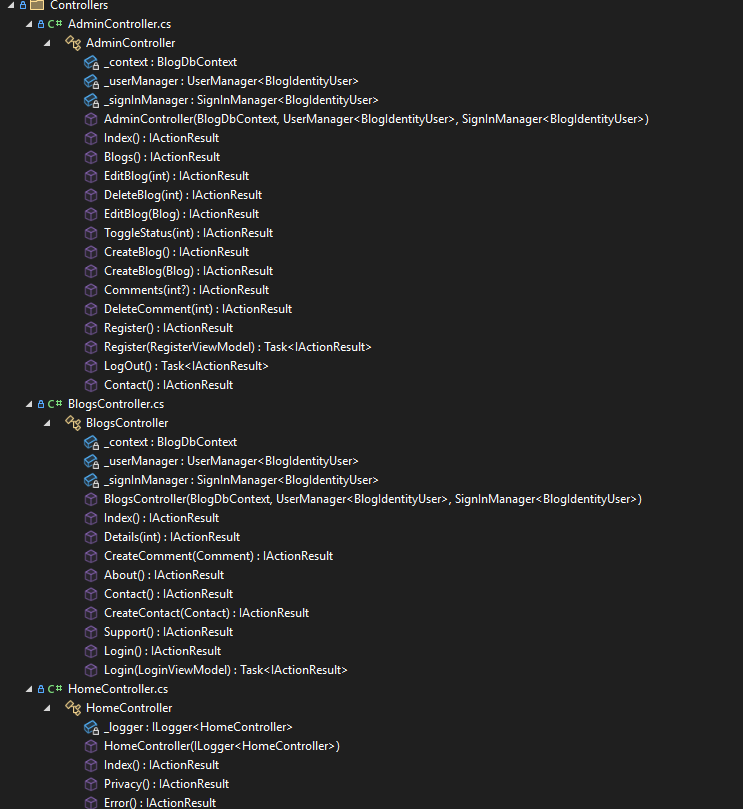
DbSet<T> genel bir türdür. T, buraya hangi model sınıfını koyarsanız, o tabloyu temsil eder.

DbSet<Blog> ise, veritabanındaki **Blogs tablosunu** temsil eden özel bir türdür.

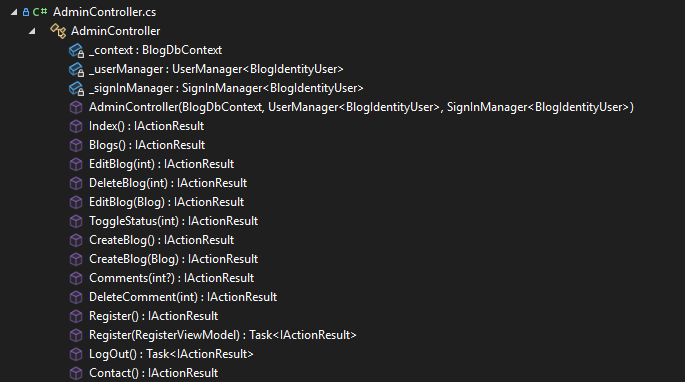
**{ get; set; }** ifadesi, bir özelliğin hem okunabilir hem de yazılabilir olduğunu belirtir ve bu işlemleri bizim için otomatik olarak yönetir

# Controllers





## **AdminController**



### **using System.Threading.Tasks;**

* **Ne yapıyor:** Bu satır, asenkron programlama için gerekli olan **Task** sınıfını kullanabilmenizi sağlar. Task, bir görevin arka planda çalışmasını ve sonucunu beklemesini yönetmek için kullanılır. async ve await anahtar kelimeleriyle birlikte kullanılır.

### **using blogv1.Context;**

### **using blogv1.Identity;**

### **using blogv1.Models;**

### **using blogv1.Models.ViewModels;**

* **Ne yapıyor:** Bunlar, projenin farklı katmanlarından (katman: bir uygulamanın birbiriyle ilgili kısımlarının oluşturduğu mantıksal bölümler, ör: veri katmanı, sunum katmanı vb.) sınıfları ve modelleri (Blog, Comment, Contact, DashboardViewModel vb.) kullanabilmek için gerekli olan referanslardır.

### **using Microsoft.AspNetCore.Authorization;**

### **using Microsoft.AspNetCore.Identity;**

### **using Microsoft.AspNetCore.Mvc;**

* **Ne yapıyor:** Bunlar, ASP.NET Core framework'ünün sağladığı temel kütüphaneleri içeri aktarır. **Authorization** (yetkilendirme), **Identity** (kimlik doğrulama) ve **Mvc** (Model-View-Controller) yapılarını kullanmak için gereklidirler.

### **namespace blogv1.Controllers**

* **Ne yapıyor:** Bu, kodun bir ad alanına (namespace) ait olduğunu belirtir. Bu, benzer işlevlere sahip sınıfları gruplamak ve isim çakışmalarını önlemek için kullanılır.

### **[Authorize]**

* **Ne yapıyor:** Bu bir niteliktir (attribute). AdminController sınıfının tamamına uygulanır. Bu, bu denetleyici içindeki her bir metodun çalıştırılmadan önce, kullanıcının sisteme giriş yapmış (yani kimliği doğrulanmış) olup olmadığını kontrol etmesini sağlar. Giriş yapmamış bir kullanıcı bu sayfalara erişemez.

### **public class AdminController : Controller**

* **Ne yapıyor:** AdminController adında bir sınıf tanımlar ve bu sınıfın ASP.NET Core MVC'deki temel Controller sınıfından miras aldığını belirtir. Bu sayede, denetleyiciye özel tüm işlevlere (örneğin View() ve RedirectToAction() gibi metotlara) erişim sağlar.

#### **private readonly BlogDbContext \_context;**

* **Ne yapıyor?** Bu değişken, uygulamanın veritabanı ile iletişim kurmasını sağlayan bir köprüdür. Bir veritabanı bağlamı (context) olarak, **Entity Framework Core**'un temelidir.
* **Kullanım Amacı:** Bu değişken sayesinde veritabanına sorgu gönderebilir, yeni veriler ekleyebilir, mevcut verileri güncelleyebilir veya silebilirsiniz. Örneğin, bir blog yazısı kaydetmek veya mevcut yazıları listelemek için \_context kullanılır.
* **readonly ve private Nedir?**
  + private: Bu değişkenin sadece tanımlandığı sınıf içinde erişilebilir olduğunu belirtir. Dışarıdan doğrudan değiştirilemez veya okunamaz. Bu, kapsülleme (encapsulation) ilkesinin bir parçasıdır ve kodun daha güvenli ve düzenli olmasını sağlar.
  + readonly: Bu değişkenin değerinin sadece tanımlandığı anda (veya yapıcı metot içinde) bir kez atanabileceğini gösterir. Değer bir kere atandıktan sonra bir daha değiştirilemez. Bu, verinin bütünlüğünü korumaya yardımcı olur.

#### **private readonly UserManager<BlogIdentityUser> \_userManager;**

* **Ne yapıyor?** Bu, ASP.NET Core Identity'nin bir parçasıdır ve kullanıcı hesaplarıyla ilgili işlemleri yönetir.
* **Kullanım Amacı:**
  + Yeni bir kullanıcı kaydetme (CreateAsync).
  + Kullanıcının şifresini değiştirme veya sıfırlama.
  + Bir kullanıcıyı role atama veya rollerini kontrol etme.
  + Kullanıcı bilgilerini bulma veya doğrulama.

#### **private readonly SignInManager<BlogIdentityUser> \_signInManager;**

* **Ne yapıyor?** Bu da ASP.NET Core Identity'nin bir parçasıdır ve kullanıcıların uygulamaya giriş-çıkış (oturum açma/kapatma) işlemlerini yönetir.
* **Kullanım Amacı:**
  + Kullanıcının girdiği e-posta ve şifrenin doğruluğunu kontrol edip oturum açma (PasswordSignInAsync).
  + Kullanıcının oturumunu kapatma (SignOutAsync).
  + İki faktörlü kimlik doğrulama gibi gelişmiş oturum yönetimi senaryolarını ele alma.

Bu üç değişken, genellikle bir yapıcı metot (constructor) aracılığıyla "Dependency Injection" (Bağımlılık Enjeksiyonu) adı verilen bir mimari desen kullanılarak sınıfınıza sağlanır. Böylece, ilgili servisleri manuel olarak oluşturmak zorunda kalmazsınız ve kodunuz daha test edilebilir ve esnek hale gelir.



### **AdminController Yapıcı Metodu (Constructor)**

Verdiğiniz kod bloğu, **Dependency Injection (Bağımlılık Enjeksiyonu)** adı verilen temel bir yazılım tasarım desenini gösteriyor. Bu yöntem, bir sınıfın (burada AdminController) ihtiyaç duyduğu diğer nesneleri (bağımlılıkları) kendi içinde oluşturmak yerine, dışarıdan almasını sağlar. Bu, kodun daha esnek, bakımı kolay ve test edilebilir olmasını sağlar.

Bu kod satırını üç ana başlık altında inceleyebiliriz:

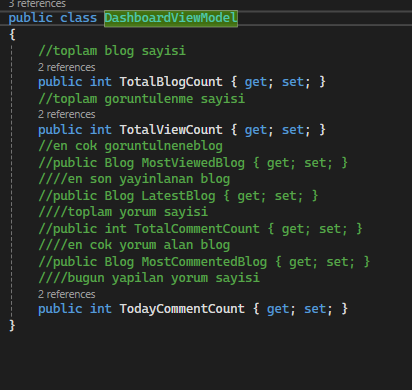
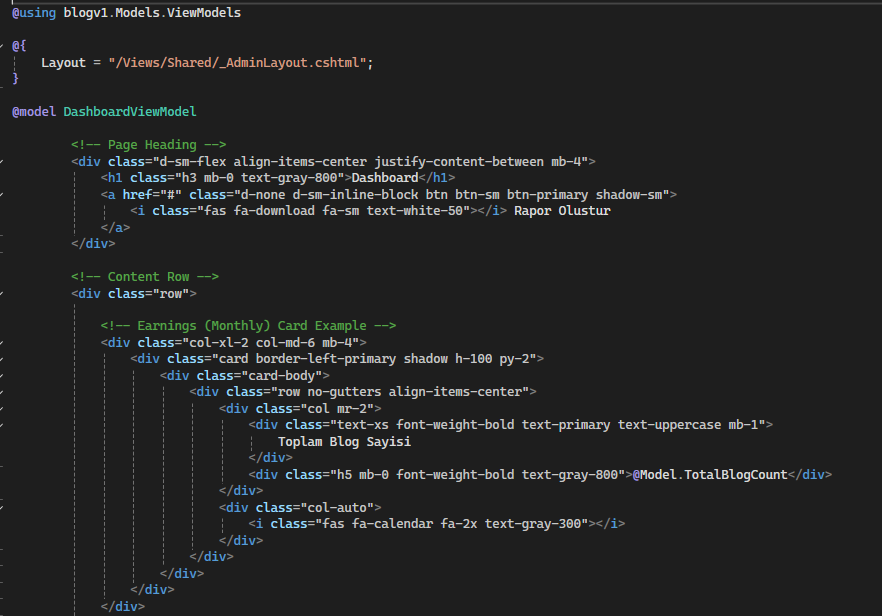
#### **1. Metot Tanımı: public AdminController(...)**

* **public**: Bu anahtar kelime, yapıcı metodun herkese açık olduğunu, yani uygulamanın herhangi bir yerinden çağrılabileceğini belirtir.
* **AdminController**: Bu, metodun ait olduğu sınıfın adıdır. Yapıcı metotların her zaman ait oldukları sınıf ile aynı adı taşıması gerekir.

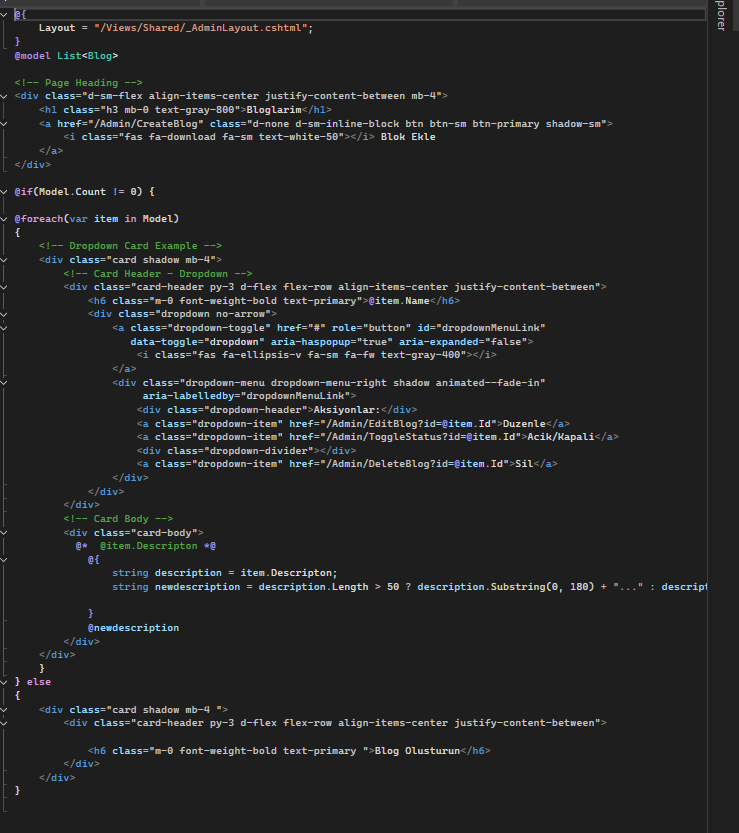
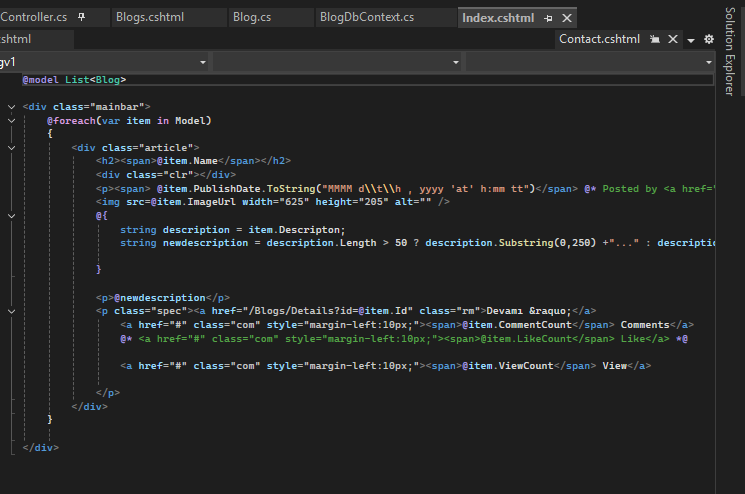
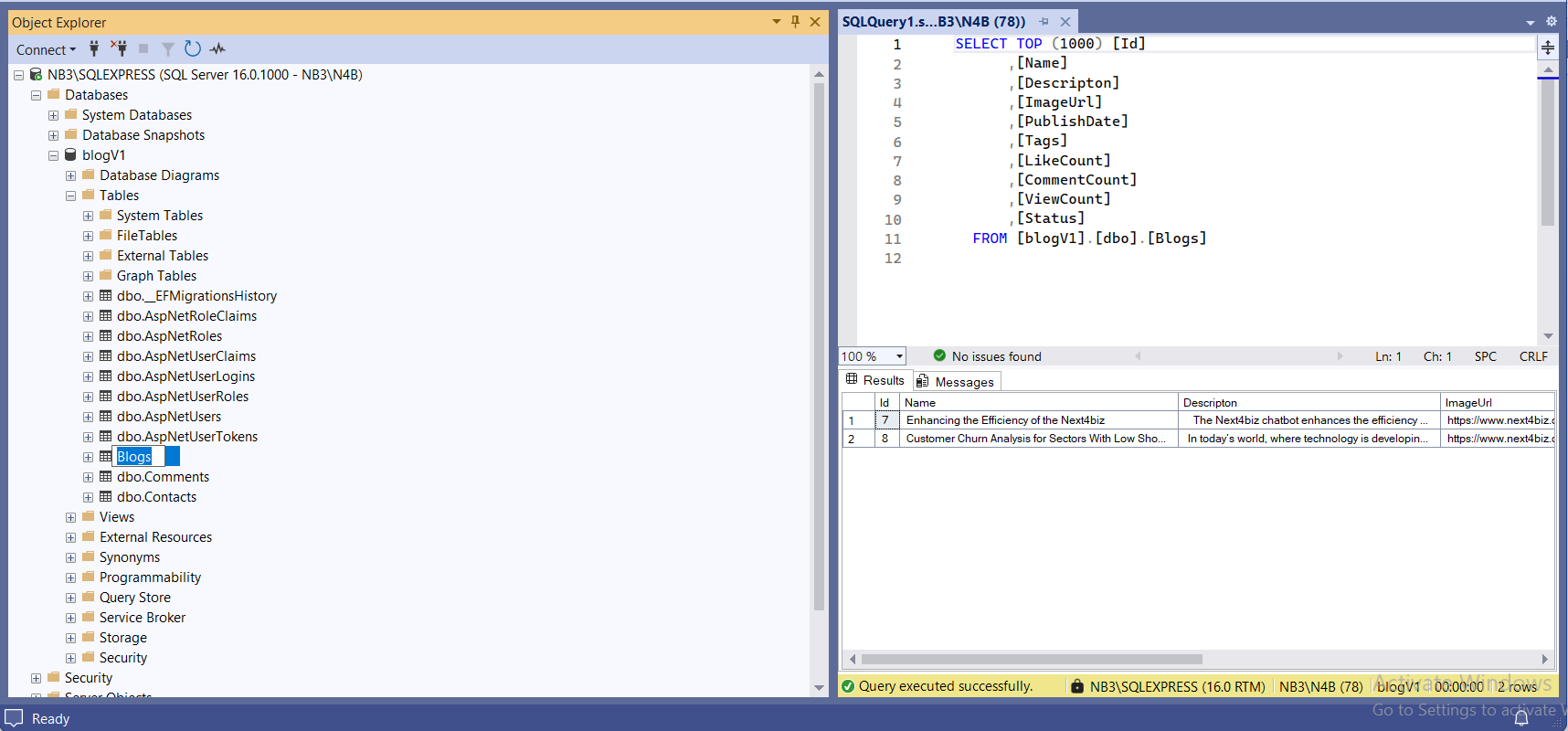
#### **2. Parametreler: (BlogDbContext context, UserManager<BlogIdentityUser> userManager, SignInManager<BlogIdentityUser> signInManager)**

* Bu parantezler içinde yer alan üç parametre, AdminController sınıfının çalışmak için ihtiyaç duyduğu servislerdir.
* **BlogDbContext context**: Veritabanı işlemlerini yöneten servis.
* **UserManager<BlogIdentityUser> userManager**: Kullanıcı hesaplarının (kayıt, şifre değişikliği vb.) yönetilmesini sağlayan servis.
* **SignInManager<BlogIdentityUser> signInManager**: Kullanıcıların oturum açma ve kapatma işlemlerini yöneten servis.

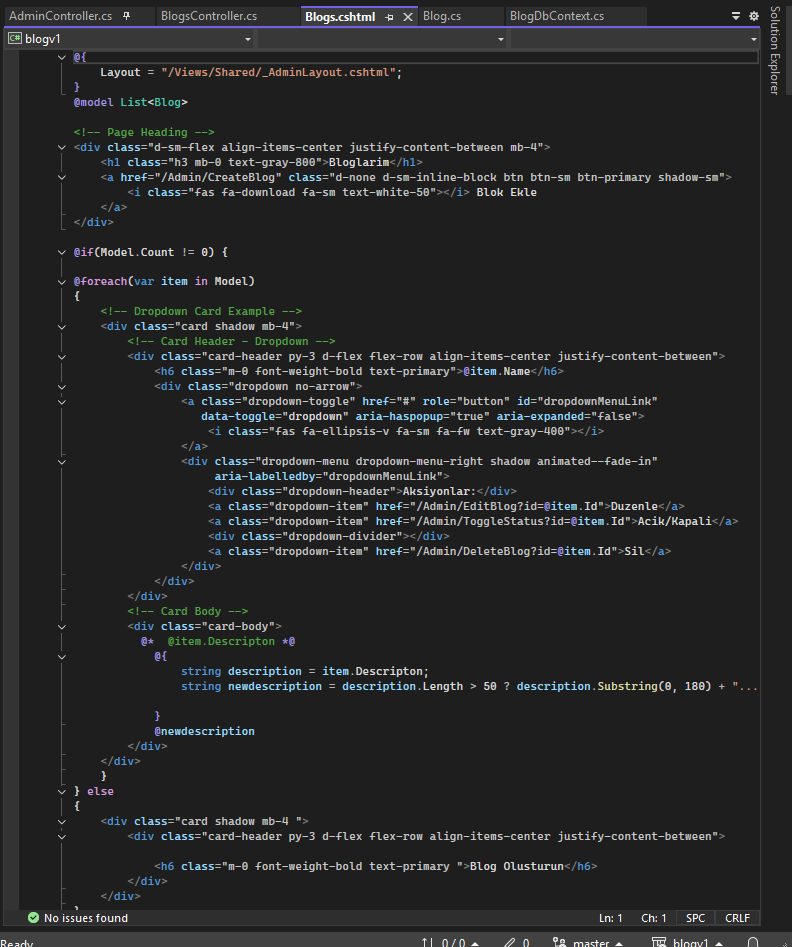
### **public IActionResult Index()**

* **var dashboard = new DashboardViewModel(); Tam Olarak Ne Oluyor?**
* Bu satır, bellekte yeni bir "kutucuk" oluşturup ona "dashboard" ismini veriyor ve bu kutucuğun içine DashboardViewModel sınıfına ait bir nesne yerleştiriyor.
* **new DashboardViewModel()**: Bu kısım, **DashboardViewModel** adında bir sınıfın yeni bir örneğini (instance) oluşturur. Sınıfları bir nevi plan veya kalıp gibi düşünebilirsiniz. new anahtar kelimesi, bu kalıba uygun, somut bir nesne yaratır. Bu nesne, DashboardViewModel sınıfının içinde tanımlanmış tüm özelliklere (properties) ve metotlara sahip olur.
* **var dashboard =**: Bu kısım, oluşturulan bu yeni nesneyi saklamak için bir değişken tanımlar.
  + **var**: Bu anahtar kelime, C# dilinde tip çıkarımı (type inference) için kullanılır. Derleyici, new DashboardViewModel() ifadesinin döndürdüğü nesnenin tipini otomatik olarak anlar ve dashboard değişkenini DashboardViewModel tipinde tanımlar. Yani, bu satır aslında DashboardViewModel dashboard = new DashboardViewModel(); ile aynı anlama gelir. Bu, kodu daha kısa ve okunabilir hale getirir.
  + **dashboard**: Bu, oluşturulan nesneye verdiğimiz isimdir. Artık kodun geri kalanında bu isimle bu nesneye erişebilir, onun özelliklerini (örneğin, dashboard.KullaniciAdi) veya metotlarını (dashboard.VeriGetir()) kullanabiliriz.
* -
* ,
* Burdan da gorundugu uzere model aslında bır sınıf oluyor
* var toplamblogsayisi = \_context.Blogs.Count(); işlemi yapıyoruz
* dashboard.TotalBlogCount = toplamblogsayisi; işlem sonucunu dashboarddaki iteme eşitliyoruz
* **return View(dashboard);**: Bu satır, **dashboard** nesnesini View (görünüm) adında bir web sayfasına gönderir. Bu View genellikle Index.cshtml gibi bir Razor sayfasıdır ve bu sayfadaki HTML kodları, gönderilen dashboard modelindeki verilere erişerek onları ekranda gösterebilir. Örneğin, toplam blog sayısını **@Model.TotalBlogCount** şeklinde kullanarak ekrana yazdırabilir. Çok mantıklı
* 

### **public IActionResult Blogs()**

* **Ne yapıyor:** Blogs eylemini tanımlar. \_context.Blogs.ToList() komutunu kullanarak veritabanındaki tüm blog gönderilerini bir liste olarak çeker ve bu listeyi **Blogs.cshtml** adlı bir görüntüleme (view) sayfasına gönderir.
* **.Blogs**: Bu, BlogDbContext içinde tanımlanmış olan ve veritabanındaki blog yazılarını temsil eden bir koleksiyondur (genellikle bir DbSet).
* **.ToList()**: Bu metot, **LINQ (Language-Integrated Query)** sorgusunu çalıştırır. \_context.Blogs ifadesi, henüz veritabanına bir sorgu göndermez, sadece bir sorgu tanımı oluşturur. .ToList() metodu çağrıldığında, sorgu veritabanına gönderilir ve tüm blog yazıları alınarak bir liste (List<Blog>) haline dönüştürülür. Bu liste, blogs adlı değişkene atanır.
* Koddaki yorum satırı, bu işlemin amacını özetliyor: //sorguya gerek yok tum blogları gosterıcem. Bu, herhangi bir filtreleme (Where) veya sıralama (OrderBy) işlemi yapılmadan doğrudan tüm blogların listeleneceğini belirtir.
* 
* Tam dedıgı gıbı bız burad lısteleme yapıyoruz sonrasın vıew ıle modele gonderıp orda lıste hazır hale getırıp goreach ıle hepsını basıyoruz. Bu usttekı admın tarafı bu alttakı de user tarafı bak 
* Gene bır model ve foreach
* **Model Tanımı: @model List<Blog>**
* Bu satır, bu Razor sayfasının, **List<Blog>** tipinde bir veri modeli beklediğini belirtir. Yani, sayfa yüklendiğinde, Controller'dan blog listesi gelmesini bekler. Bu, sayfadaki verilere @Model anahtar kelimesi ile erişilmesini sağlar.
* **Döngü ve Blog Verilerini İşleme: @foreach(var item in Model)**
* **@foreach**: Bu, C# dilinde bir döngüdür ve Razor sayfalarında HTML içinde C# kodu çalıştırmak için @ sembolü ile kullanılır. Bu döngü, Model olarak gelen her bir Blog nesnesi için aşağıdaki HTML bloğunu tekrarlar.
* **var item in Model**: Döngü her döndüğünde, mevcut blog nesnesi item değişkenine atanır.
* Controllerda buna ulasıyor

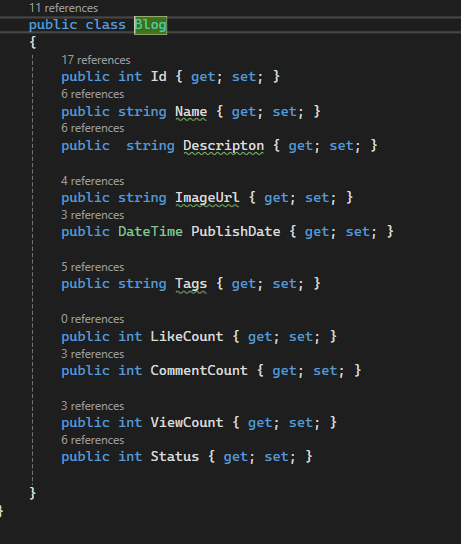
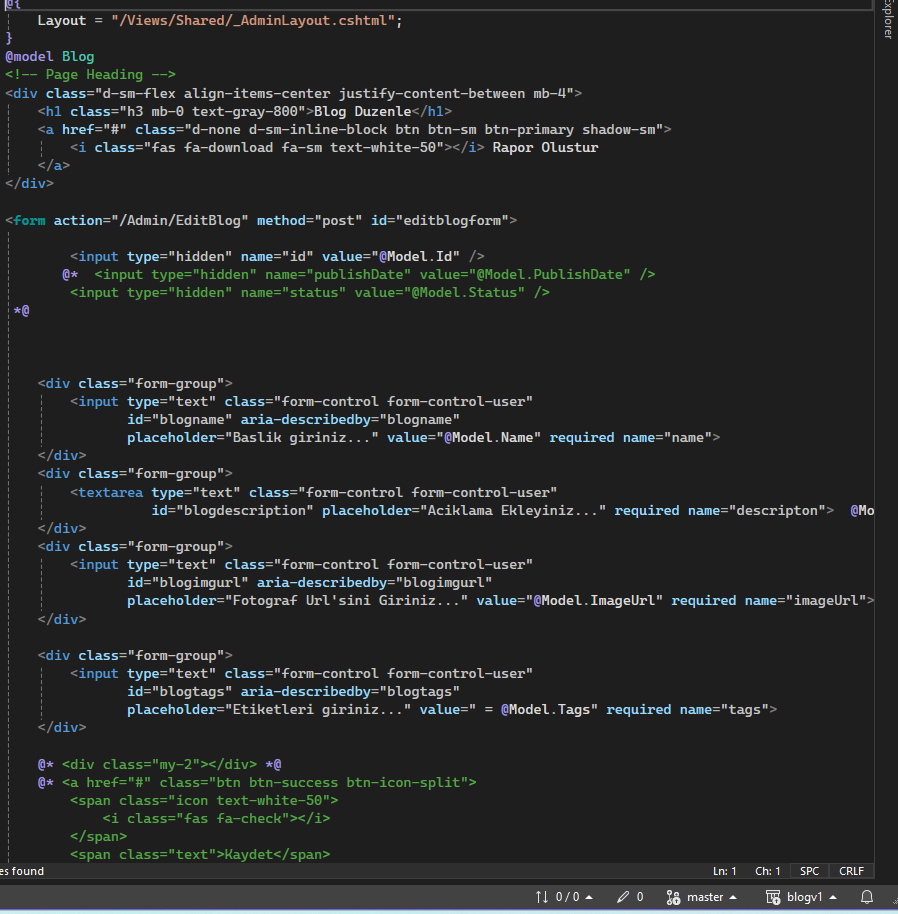
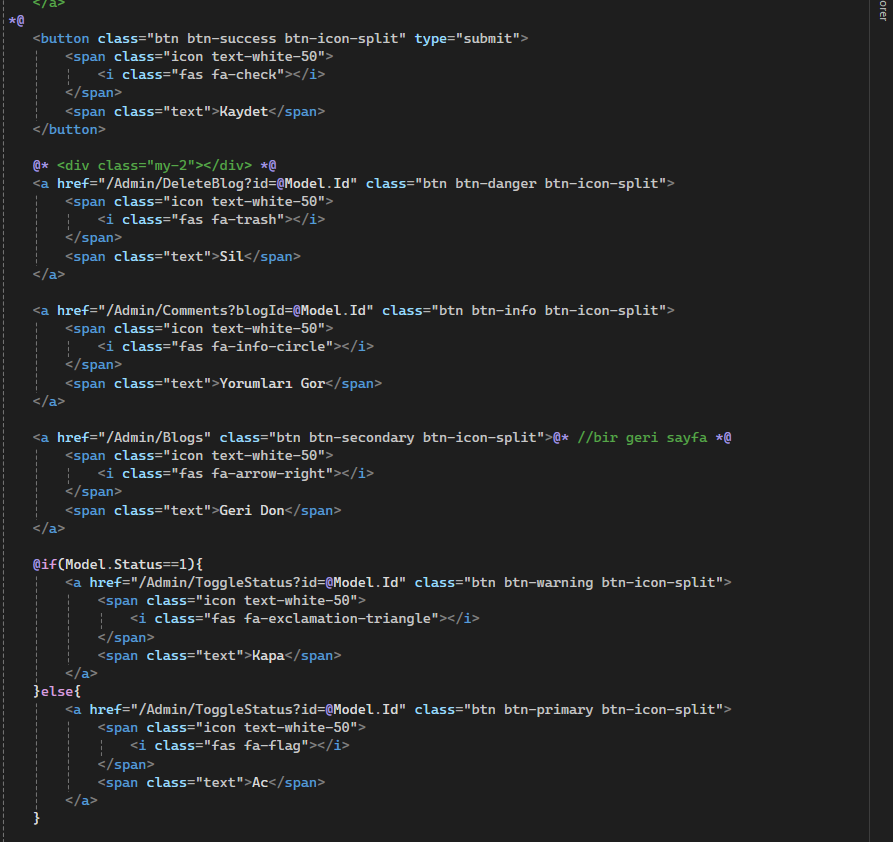
### **public IActionResult DeleteBlog(int id)**

* **Blog ID'sini Alma**
* **public IActionResult DeleteBlog(int id)**: Metot, id adında bir **integer (tamsayı)** parametre alır. Bu ID, silinecek blog yazısını benzersiz bir şekilde tanımlar.
* **Silinecek Blogu Bulma**
* **var blog = \_context.Blogs.Where(x => x.Id == id).FirstOrDefault();**:
  + **\_context.Blogs**: Veritabanındaki Blogs tablosuna erişim sağlar.
  + **.Where(x => x.Id == id)**: Bu **LINQ** sorgusu, Blogs tablosundaki tüm kayıtları ID'lerine göre filtreler. Yalnızca gelen id parametresiyle eşleşen blogları seçer.
  + **.FirstOrDefault()**: Filtreleme sonucunda bulunan ilk kaydı alır. Eğer eşleşen bir blog bulunamazsa, null döner. Bu adım, veritabanından yalnızca bir tane blog kaydı getirmek için kullanılır.
* **Blogu Silme Komutunu Verme**
* **\_context.Blogs.Remove(blog);**: Bu satır, Remove metodunu kullanarak, bellekte bulunan blog nesnesini veritabanından silinmek üzere işaretler. Bu noktada blog henüz veritabanından silinmemiştir, sadece **Entity Framework** tarafından "silinecek" olarak etiketlenmiştir.
* **Değişiklikleri Veritabanına Kaydetme**
* **\_context.SaveChanges();**: Bu metot çağrıldığında, Entity Framework, işaretlenmiş olan tüm değişiklikleri (silme, ekleme, güncelleme) toplar ve tek bir işlemde veritabanına göndererek kalıcı hale getirir. Bu satırdan sonra, ilgili blog kaydı veritabanından silinmiş olur.
* return RedirectToAction("Blogs"); blogs burda sitesel olarak gıdecegı yer ondan metın oalrak alıyor
* tekrar bu sayfayı yukluyor yani blogların oldugu edıt kısmını

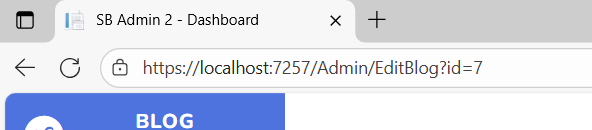
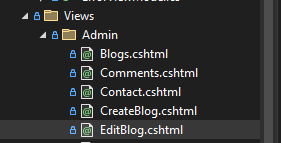
### **[HttpPost]**

* **Ne yapıyor:** Bu nitelik, bir metodun sadece HTTP POST isteği ile çalışabileceğini belirtir. Genellikle form gönderimlerinde kullanılır. EditBlog ve CreateBlog metotları için, kullanıcının bir form gönderdiğini anlar ve bu metotları çalıştırır.

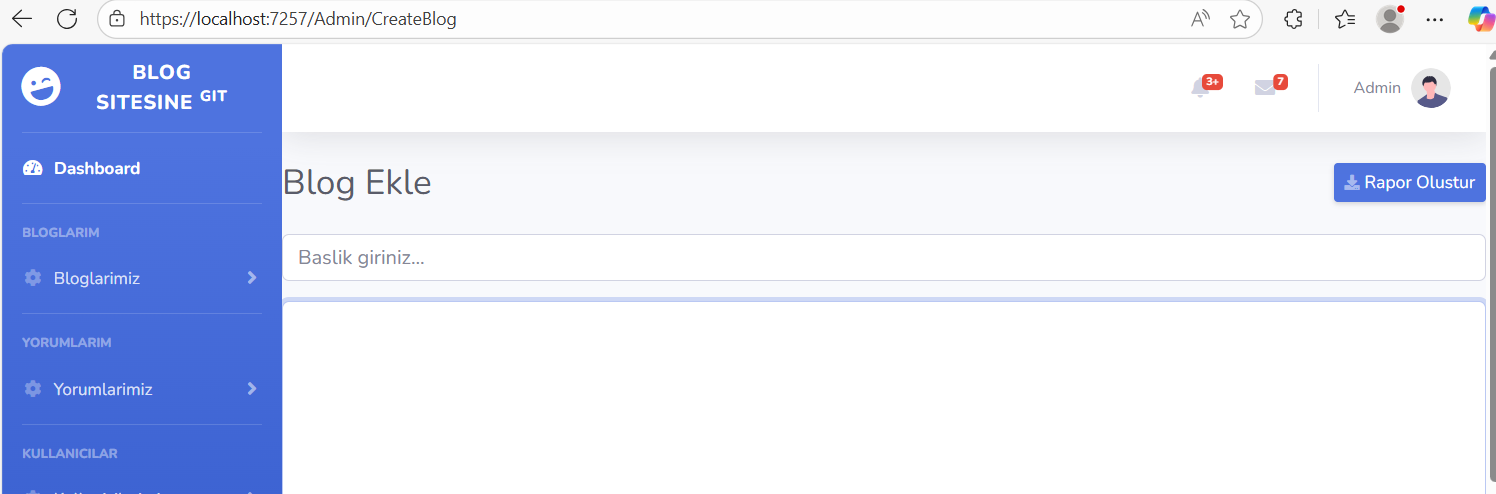
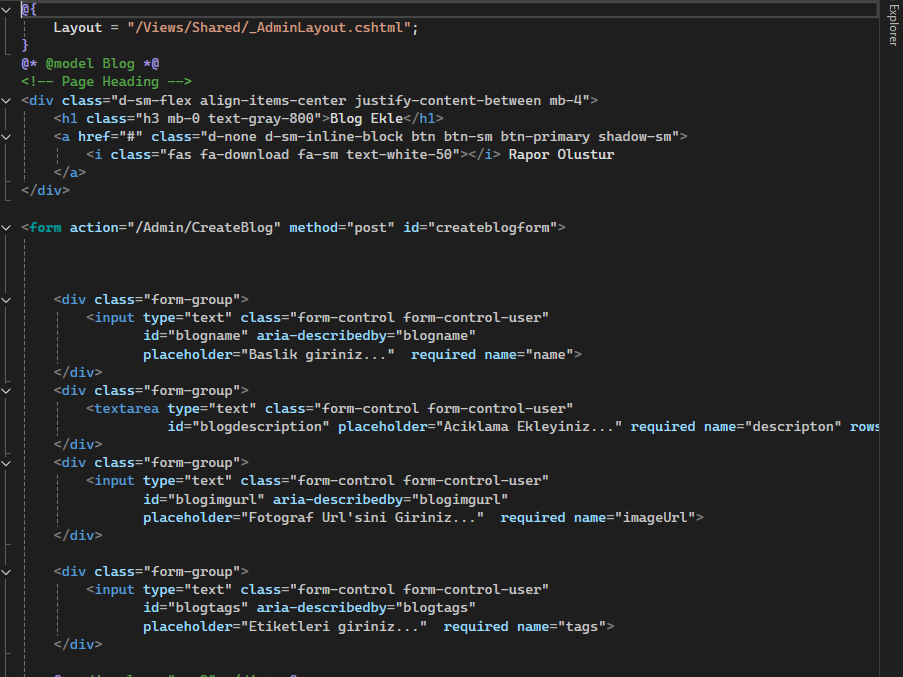
### **public IActionResult EditBlog(Blog model)**

* **public IActionResult EditBlog(Blog model)**: Bu metot, Blog türünde bir **model** parametresi alır. Bu model, kullanıcının formda doldurduğu ve sunucuya gönderdiği güncel blog verilerini (başlık, açıklama, etiketler, görsel URL'si vb.) içerir
* 
* **Mevcut Blogu Veritabanından Bulma**
* **var blog = \_context.Blogs.Where(x => x.Id == model.Id).FirstOrDefault();**:
  + Bu satır, model nesnesinin Id özelliğini kullanarak, veritabanındaki mevcut blog kaydını bulur. Bu adım çok önemlidir çünkü doğrudan gelen model nesnesini güncellemek yerine, veritabanındaki **orijinal** nesne üzerinde değişiklik yapılır. Bu, Entity Framework'ün değişiklikleri doğru bir şekilde takip etmesini sağlar.
  + Bu noktada, bellekte iki farklı blog nesnesi vardır: veritabanından gelen **blog** nesnesi ve kullanıcının formdan gönderdiği **model** nesnesi.
* **Özellikleri Güncelleme**
* **blog.Name = model.Name;**: Orijinal blog nesnesinin Name özelliği, formdan gelen model nesnesinin Name özelliği ile güncellenir.
* **blog.Descripton = model.Descripton;**: Açıklama (Descripton) özelliği güncellenir.
* **blog.Tags = model.Tags;**: Etiketler (Tags) özelliği güncellenir.
* **blog.ImageUrl = model.ImageUrl;**: Görsel URL'si (ImageUrl) özelliği güncellenir.
* **Değişiklikleri Kaydetme ve Yönlendirme**
* **\_context.SaveChanges();**: Bu metot, yapılan tüm değişiklikleri veritabanına kalıcı olarak kaydeder.
* **return RedirectToAction("Blogs");**: Güncelleme işlemi tamamlandıktan sonra, kullanıcıyı tekrar blog listeleme sayfasına yönlendirir. Bu, kullanıcının düzenlediği blogun güncel halini listede görmesini sağlar. Bu yönlendirme, GET isteğiyle çalışan Blogs eylemini tetikler ve tarayıcının sayfayı yeniden yüklemesini sağlar.
* 
* Blog duzenleme sayfasına boyle verıyor modelle
* 

### **public IActionResult ToggleStatus(int id)**

* Bu metot, bir blog yazısının yayımlanma durumunu (aktif/pasif) değiştirir.
* Şu yukardaki if else dongusu aslında yani orda kullanılıyor
* **<a href="/Admin/ToggleStatus?id=@Model.Id" ... >**:
* Bu bir HTML bağlantı etiketidir.
* **href**: Butonun tıklanıldığında gideceği adresi belirtir.
* **"**[**/Admin/ToggleStatus?id=@Model.Id**](mailto:/Admin/ToggleStatus?id=@Model.Id)**"**: Bu adres, Admin kontrolcüsünün içindeki ToggleStatus eylemini çağırır ve bu eyleme, blogun benzersiz kimliğini (Model.Id) bir parametre olarak gönderir.
* 
* 
* Bak bunlar burdan dırekt usttekı lınkı alıyor

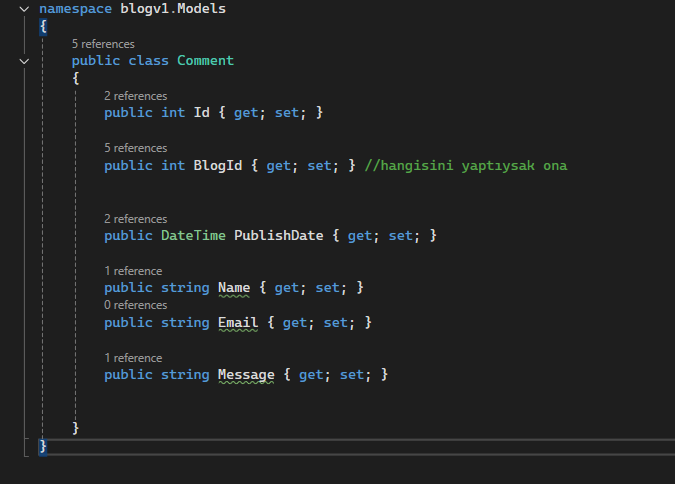
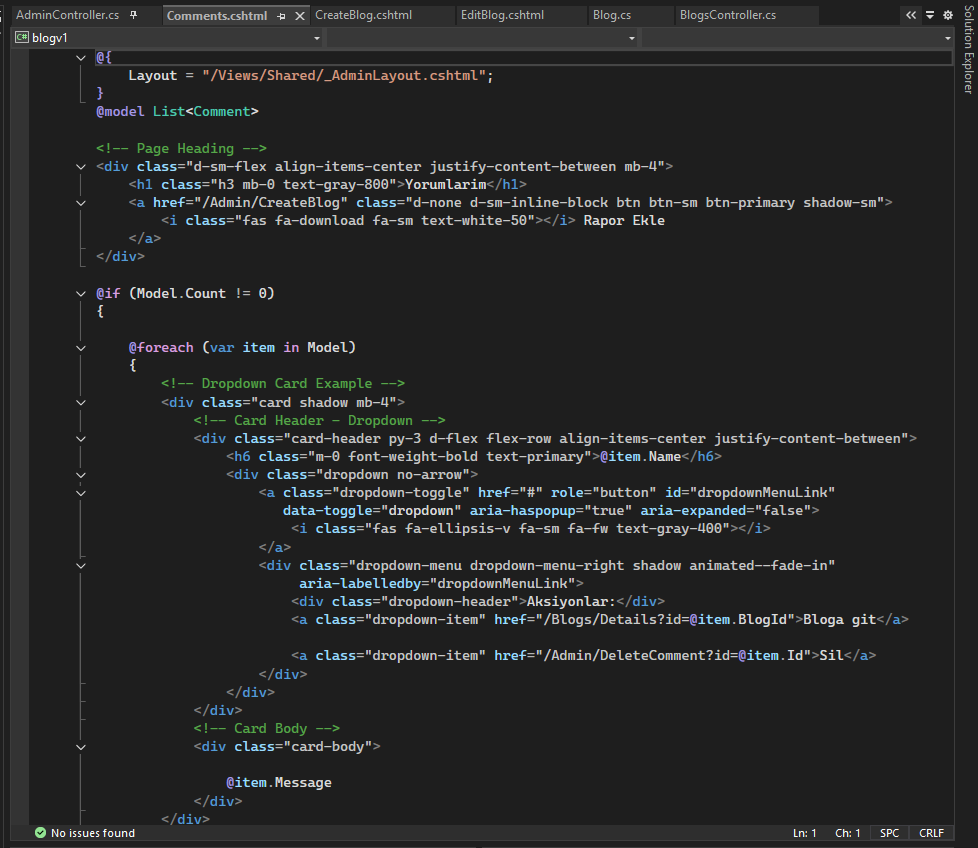
### **public IActionResult CreateBlog()**

* **GET metodu** boş bir blog oluşturma formunu gösterir. Alttakını yani
* 
* 

### **public IActionResult CreateBlog(Blog model)**

* Bu eylem, kullanıcı formu doldurup "Kaydet" veya "Oluştur" butonuna tıkladığında çalışır.
* **[HttpPost]**: Bu etiket, bu metodun yalnızca **HTTP POST** istekleri tarafından çağrılabileceğini belirtir. Bu, veritabanına veri gönderen işlemler için güvenlik ve doğruluk sağlar.
* **Blog model**: Bu, formdan gelen verilerin otomatik olarak Blog nesnesine bağlanmasını (Model Binding) sağlar. Kullanıcının formda girdiği başlık, içerik, etiketler gibi bilgiler bu model nesnesine aktarılır.
* **model.PublishDate = DateTime.Now;**: Blogun PublishDate (yayınlanma tarihi) özelliği, işlemin yapıldığı anın tarihi ve saati ile doldurulur. Bu, kullanıcının girmesi gerekmeyen ancak sistemin otomatik olarak eklemesi gereken bir veridir.
* **model.Status = 1;**: Blogun Status (durum) özelliği **1** olarak ayarlanır. 1 genellikle blogun "aktif" veya "yayınlanmış" olduğunu belirtir.
* **\_context.Blogs.Add(model);**: Bu komut, hazırlanan model nesnesini veritabanındaki Blogs koleksiyonuna eklemek için Entity Framework'e bildirir. Bu aşamada, veritabanında henüz bir değişiklik yapılmaz, yalnızca yeni bir kayıt için hazırlık yapılır.
* **\_context.SaveChanges();**: Bu komut, Entity Framework'e, bekleyen tüm değişiklikleri (bu durumda yeni blog ekleme) veritabanına kalıcı olarak kaydetmesi talimatını verir. Bu adımdan sonra, yeni blog yazısı veritabanına kaydedilmiş olur.
* **return RedirectToAction("Blogs");**: İşlem tamamlandıktan sonra, kullanıcı, blog listeleme sayfasını gösteren Blogs eylemine yönlendirilir. Bu, kullanıcının yeni eklediği blogu listede görebilmesini sağlar.
* Baya iyi

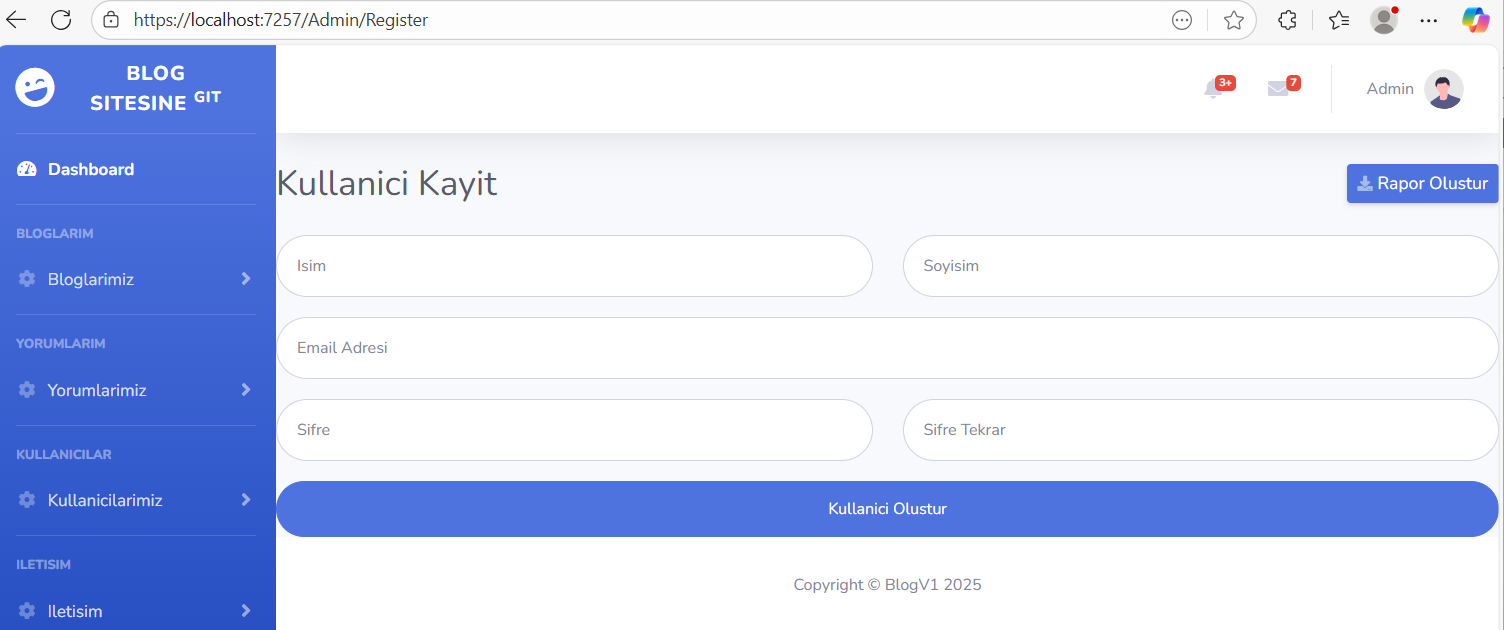
### **public IActionResult Comments(int? blogId)**

* blogId parametresi varsa o bloga ait yorumları, yoksa tüm yorumları gösterir.
* **?**: Soru işareti, bu tamsayının **null** olabileceğini, yani parametrenin opsiyonel olduğunu gösterir.
* **var comments = new List<Comment>();**
* **var**: C# dilinde değişken tipini derleyicinin otomatik olarak belirlemesini sağlar.
* **comments**: Değişkenin adıdır. Bu değişken, veritabanından çekilecek yorumları tutacaktır.
* **new List<Comment>()**: Comment nesnelerinden oluşacak boş bir liste oluşturur. Bu, değişkeni başlangıçta boş bir liste olarak başlatır. Kodun içindeki yorum satırı (hpsterıyor nereden geldıgını bu sefer 0 degılse) bu satırın amacını net olmasa da belirtiyor: Yorum listesinin başlangıç noktasını oluşturmak.
* **return View(comments);**
* **return View(...)**: Bu metot, hazırlanan comments listesini bir web sayfasına (View) gönderir.
* **View(comments)**: Otomatik olarak, metot adıyla eşleşen Comments.cshtml adındaki View dosyasını bulur ve comments listesini bu View'a bir model olarak gönderir. View dosyasında bu listeye @Model anahtar kelimesi ile erişilebilir.
* 
* O gonderılenı boyle de vıew de kullnaıyoruz.
* Varsayılan olarak, ASP.NET Core, View dosyalarını bulmak için belirli bir yolu takip eder: [Kontrolcü Adı]/[Eylem Adı].cshtml ve paylaşılan (Shared) View'lar için Shared klasörü. Bu yol, Views klasörünün içinde aranır.
* Eğer klasör adınızı Views yerine Goruntu veya başka bir şey yaparsanız, uygulamanıza View'ları nerede araması gerektiğini söylemeniz gerekir.

### **public IActionResult DeleteComment(int id)**

* Belirli bir yorumu veritabanından siler.

### **public IActionResult Register()**

* Register sayfasını gosterır

### **Neden IActionResult Kullanıyoruz?**

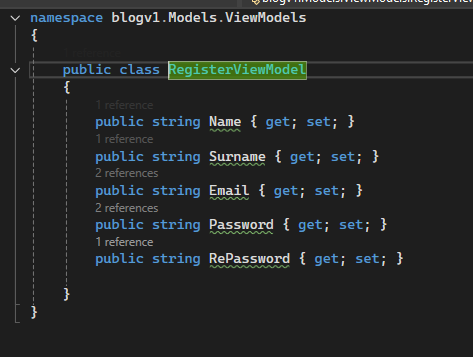
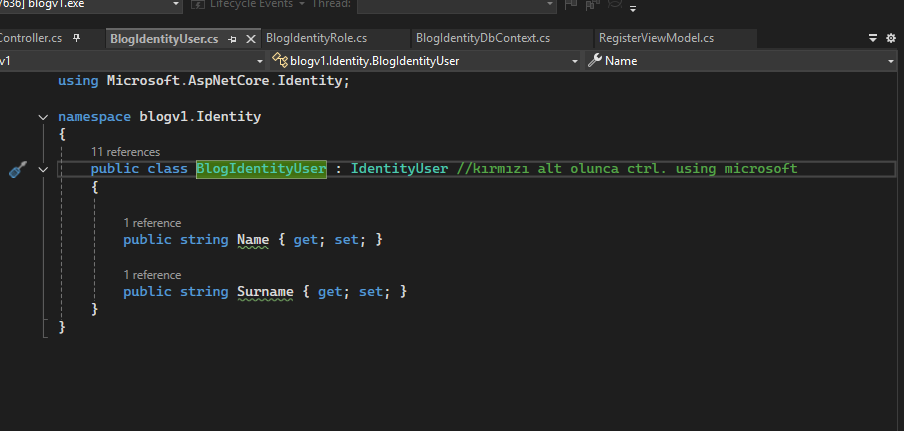
#### **1. Esneklik ve Farklı Sonuç Tipleri**

Bir kontrolcü metodu sadece bir View (sayfa) döndürmez. Farklı senaryolarda farklı sonuçlara ihtiyaç duyabilir:

* **View**: Bir web sayfası göstermek için kullanılır (örneğin, return View(model)).
* **Redirect (Yönlendirme)**: Kullanıcıyı başka bir URL'ye yönlendirmek için kullanılır (örneğin, return RedirectToAction("Blogs")).
* **JSON**: Bir API çağrısında veri döndürmek için kullanılır (örneğin, return Json(data)).
* **File (Dosya)**: Kullanıcıya bir dosya indirtmek için kullanılır.
* **NotFound**: Kaynak bulunamadığında 404 hatası döndürmek için kullanılır (örneğin, return NotFound()).
* **Ok/BadRequest**: Başarılı (200 OK) veya hatalı (400 Bad Request) bir işlem sonucunu belirtmek için kullanılır.

IActionResult bu farklı sonuç tiplerinin hepsini kapsayan bir şemsiye gibidir. Bu sayede, aynı metot içinde if veya else bloklarıyla farklı dönüş tipleri kullanabilirsiniz.

### **public async Task<IActionResult> Register(RegisterViewModel model)**

* Az oncekı logın gıbı bu da kullanıcı olustura basınca tetıklenıyor
* 
* Degıskenlerı bunlar
* Username atamak gerekıyor
* **async Task<IActionResult> Register(RegisterViewModel model)**: Metot, RegisterViewModel tipinde bir **model** parametresi alır. Bu model, kullanıcının kayıt formuna girdiği ad, soyad, e-posta ve şifre gibi bilgileri içerir. Metot, asenkron (async) çalışır ve bir sonuç (IActionResult) döndürür.
* Tanımlarız yeni
* **Kullanıcı Nesnesi Oluşturma**
* **var user = new BlogIdentityUser { ... };**: Şifreler eşleşirse, formdan gelen verilerle **BlogIdentityUser** adında yeni bir kullanıcı nesnesi oluşturulur.
  + Name, Surname, Email ve UserName gibi özellikler, model nesnesinden alınan verilerle doldurulur. ASP.NET Core Identity'de UserName genellikle Email ile aynı değere ayarlanır.
* **Kullanıcıyı Veritabanına Kaydetme**
* **var result = await \_userManager.CreateAsync(user, model.Password);**: Bu, **ASP.NET Core Identity**'nin ana bileşenlerinden biri olan \_userManager'ın CreateAsync metodunu çağırır.
  + Bu metot, user nesnesini ve düz metin şifreyi (model.Password) parametre olarak alır.
  + **CreateAsync** metodu, şifreyi otomatik olarak **hash'ler** (güvenli bir şekilde şifreler), kullanıcıyı veritabanına kaydeder ve bu işlemin sonucunu (IdentityResult) döner.
  + **await** anahtar kelimesi, işlemin asenkron olduğunu ve sonucun beklenmesi gerektiğini belirtir. Bu, uygulamanın bu işlem sırasında engellenmemesini sağlar.
* Registerda şifre Büyük harf kucuk harf sayı ve sembol ıcermesı gerekıyor kutuphaneden gelen ozellık

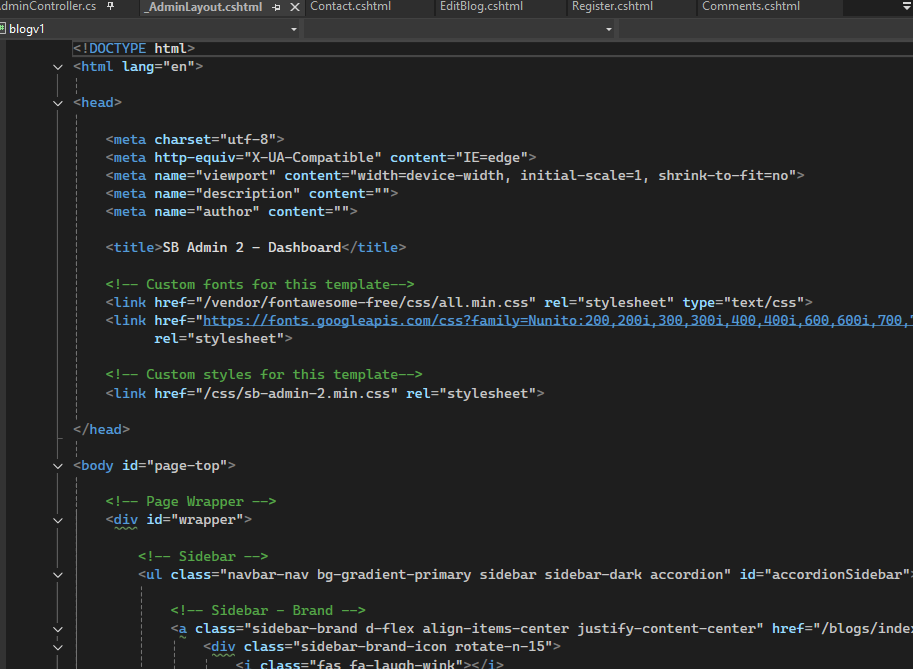
### **Kütüphaneden Gelen Şifre Kuralları**

\_userManager.CreateAsync() metodu, kullanıcının girdiği şifreyi kaydetmeden önce varsayılan olarak şu kontrolleri yapar:

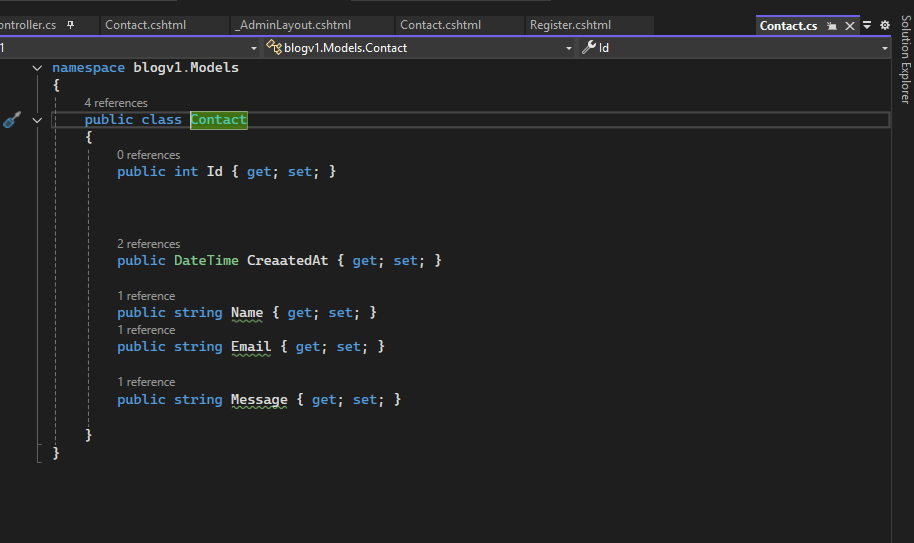
* **En az 1 büyük harf**
* **En az 1 küçük harf**
* **En az 1 sayı**
* **En az 1 özel karakter (sembol)**
* **En az 6 karakter uzunluğunda**

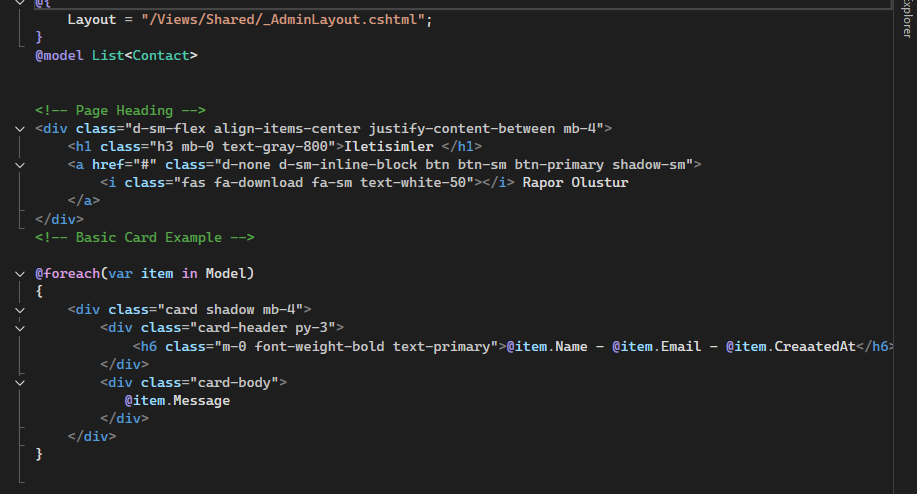
Bu kontroller, kullanıcı hesabının güvenliğini artırmak için tasarlanmıştır. CreateAsync metodu bu kurallardan herhangi biri ihlal edilirse **başarısız olur** ve result.Succeeded özelliği false değerini döner.

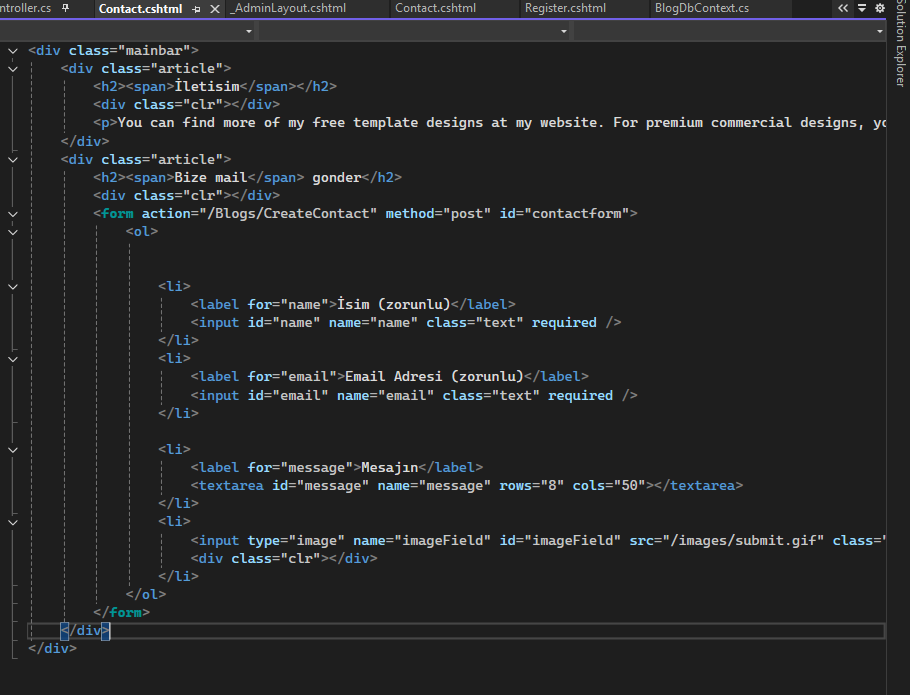
### **public async Task<IActionResult> LogOut()**

* Oturum açmış kullanıcının oturumunu güvenli bir şekilde kapatır ve ana sayfaya yönlendirir.
* Admınlayoutta bunun ıslemlerı vıew kısmı yani shared altında
* **return RedirectToAction("Index", "Blogs");**
* **return RedirectToAction(...)**: Bu komut, kullanıcıyı başka bir eyleme yönlendirir.
* **"Index"**: Yönlendirilecek eylemin adıdır. Bu durumda, Index adlı eylem.
* **"Blogs"**: Yönlendirilecek eylemin bulunduğu **kontrolcünün** adıdır.
* Bu komut, kullanıcının oturumu kapandıktan sonra **BlogsController** içindeki **Index** eylemine yönlendirileceğini belirtir.

### **public IActionResult Contact()**

* İletişim formundan gelen mesajları listeler.
* 

Boyle gosterır admınde

Kullanıcı tarafı da boyle gorunur 

### **Neden CreateContact.cshtml Yok?**

Gönderdiğiniz HTML formunun action="/Blogs/CreateContact" adresine veri gönderdiğini belirtmiştiniz. Bu, formun verileri **sunucu tarafındaki bir eyleme (action) göndereceği** anlamına geliyor. Bu eylemin bir görevi de **gelen veriyi işlemek ve veritabanına kaydetmektir.**

Bu işleme ait bir View dosyasına ihtiyacınız yoktur, çünkü bu eylem bir View döndürmez. Bunun yerine, işlem başarıyla tamamlandıktan sonra kullanıcıyı başka bir sayfaya **yönlendirmesi** beklenir.

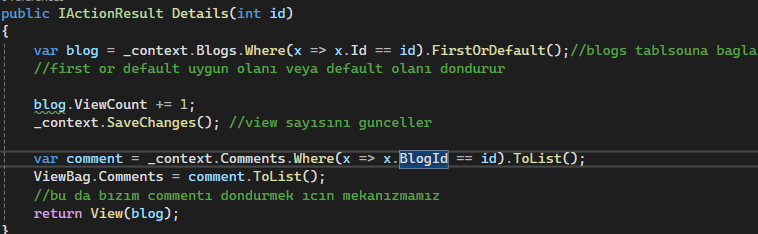
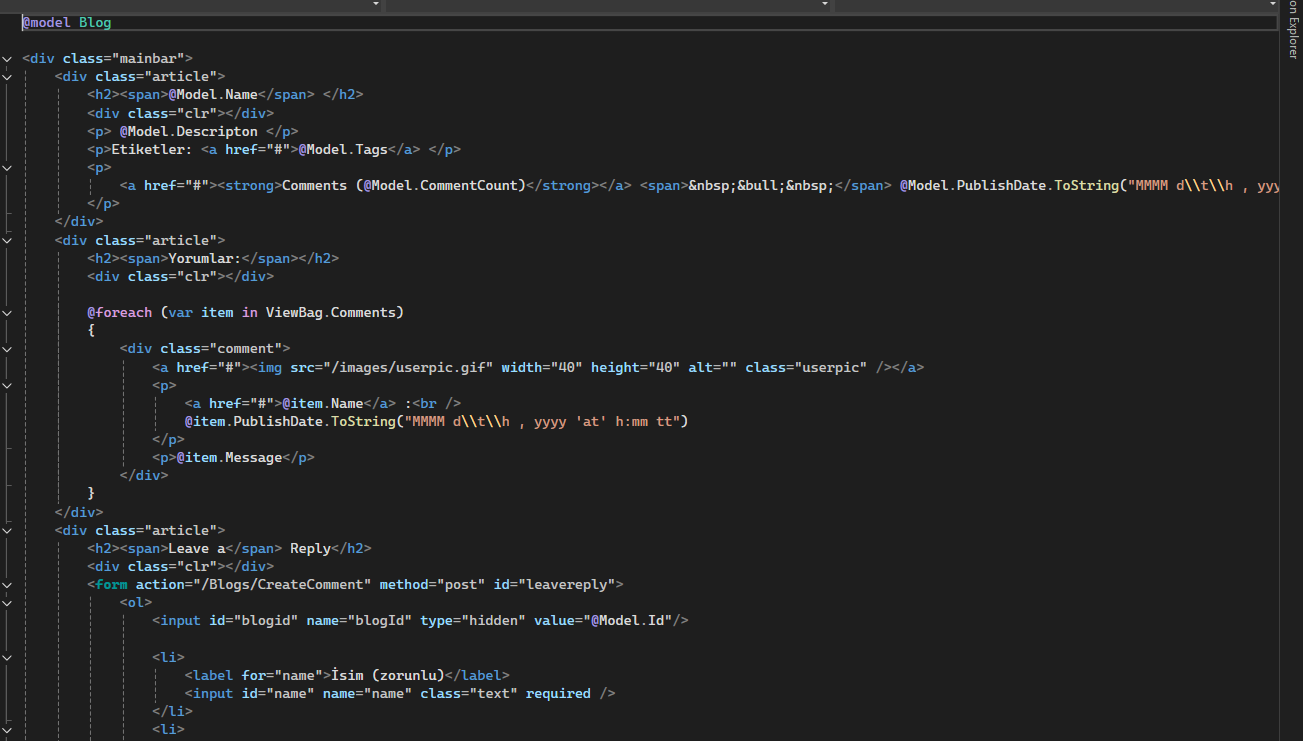
## **BlogsController**

Ilk ksıım gereklı olan verı tabanı baglantıları yapılır ne kullanınılcaksa

### **public IActionResult Index()**

* **var blogs = \_context.Blogs.Where(x => x.Status == 1).ToList();**: Bu satır, veritabanından sadece Status (durum) değeri 1 olan (yani yayınlanmış) blog yazılarını çeker. Bu, taslak halindeki blogların ana sayfada görünmesini engeller.
* **return View(blogs);**: Çektiği yayınlanmış blogların listesini, Index adlı sayfaya (View) gönderir ve kullanıcıya gösterir.
* 

### **public IActionResult Details(int id)**

* 
* **return View(blog); Satırının Anlamı**
* Bu satır, eylemin son çıktısını belirtir ve kodun kritik bir parçasıdır.
* **return View(blog);**: Bu komut, kontrolcünün işini bitirdiğini ve bir View döndüreceğini söyler. Parantez içindeki blog nesnesi, bu View'a bir **model** olarak gönderilir.
* Bu durumda, Details eylemi, adıyla aynı olan Details.cshtml adlı View dosyasını arar.
* View dosyası, **@model Blog** şeklinde tanımlanmışsa, bu sayfaya gönderilen blog nesnesine @Model anahtar kelimesiyle doğrudan erişebilir.
* 
* Detaıls cshtml de bu

### **public IActionResult CreateComment(Comment model)**

* **model.PublishDate = DateTime.Now;**: Yorumun yayınlanma tarihini o anki tarih ve saat olarak ayarlar.
* **\_context.Comments.Add(model);**: Yeni yorumu veritabanına eklenmek üzere işaretler.
* **var blog = ...; blog.CommentCount += 1;**: Yorumun ait olduğu blogu bulur ve o blogun CommentCount (yorum sayısı) değerini bir artırır.
* **\_context.SaveChanges();**: Yeni yorumu ve güncellenmiş yorum sayısını veritabanına kalıcı olarak kaydeder.
* **return RedirectToAction("Details", new {id = model.BlogId});**: İşlem tamamlandığında, kullanıcıyı yorumun yapıldığı blogun detay sayfasına yönlendirir.

### **public IActionResult About() ve public IActionResult Contact()**

Bu metotlar, sadece ilgili sayfaları (About.cshtml ve Contact.cshtml) görüntülemek için kullanılır. Veritabanıyla herhangi bir etkileşimleri yoktur.

### **public IActionResult CreateContact(Contact model)**

Bu metot, iletişim formundan gelen verileri kaydeder.

* **model.CreaatedAt = DateTime.Now;**: Mesajın oluşturulduğu tarihi ve saati belirler.
* **\_context.Contacts.Add(model);**: Mesajı veritabanına eklenmek üzere hazırlar.
* **\_context.SaveChanges();**: Değişiklikleri veritabanına kaydeder.
* **return RedirectToAction("Index");**: Mesaj gönderildikten sonra kullanıcıyı ana sayfaya yönlendirir.

### **public IActionResult Support()**

Destek sayfasını gösterir. Veritabanı işlemi içermez.

### **public IActionResult Login()**

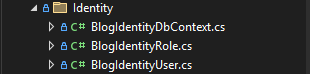
Kullanıcı giriş sayfasını gösterir.

### **[HttpPost] public async Task<IActionResult> Login(LoginViewModel model)**

Bu metot, kullanıcı giriş bilgilerini kontrol eder ve oturum açma işlemini gerçekleştirir.

* **var user = await \_userManager.FindByEmailAsync(model.Email);**: Formdan gelen e-posta adresiyle eşleşen bir kullanıcıyı asenkron olarak bulur.
* **if(user == null)**: Eğer kullanıcı bulunamazsa, tekrar giriş sayfasını gösterir.
* **var result = await \_signInManager.PasswordSignInAsync(...)**: Bulunan kullanıcının şifresini kontrol eder ve başarılıysa oturum açma işlemini gerçekleştirir.
* **if (result.Succeeded)**: Giriş başarılı olursa, kullanıcıyı Admin denetleyicisindeki Index sayfasına yönlendirir.
* **else**: Giriş başarısız olursa, tekrar giriş sayfasını gösterir.

# Identity

* 

## BlogIdentityDbContext

* using Microsoft.AspNetCore.Identity.EntityFrameworkCore;: Bu satır, IdentityDbContext sınıfını kullanabilmek için gerekli olan kütüphaneyi içeri aktarıyor. Bu kütüphane, **Entity Framework Core** ile **ASP.NET Core Identity** arasındaki entegrasyonu sağlıyor.
* using Microsoft.EntityFrameworkCore;: Bu satır, veritabanı işlemlerinin temelini oluşturan **Entity Framework Core** sınıflarını içeri aktarıyor.
* public class BlogIdentityDbContext : IdentityDbContext<BlogIdentityUser,BlogIdentityRole,string>: Bu satır, BlogIdentityDbContext adında bir sınıf tanımlıyor. Bu sınıf, IdentityDbContext sınıfından türediği için ASP.NET Core Identity'nin tüm işlevlerini miras alıyor.
* **<BlogIdentityUser, BlogIdentityRole, string>**: Bu kısım, IdentityDbContext'in hangi türde kullanıcı, rol ve anahtar (primary key) kullanacağını belirliyor.
  + BlogIdentityUser: Uygulamanın **kullanıcı modeli** (Id, Email, PasswordHash vb. özellikleri içerir).
  + BlogIdentityRole: Uygulamanın **rol modeli** (Admin, User gibi rolleri tanımlar).
  + string: Kullanıcı ve rol nesnelerinin **benzersiz kimlik (ID) alanı** için kullanılacak veri tipini belirtiyor.
* public BlogIdentityDbContext(DbContextOptions<BlogIdentityDbContext> options):base(options) { ... }: Bu sınıfın **kurucu metodudur**.
* DbContextOptions<BlogIdentityDbContext> options: Bu parametre, veritabanı bağlantı bilgilerini ve diğer yapılandırma seçeneklerini içeriyor. Bu bilgiler genellikle uygulamanın appsettings.json dosyasından okunur.
* :base(options): Gelen options parametresini, miras alınan IdentityDbContext sınıfının kurucu metoduna gönderiyor. Bu, ASP.NET Core Identity'nin veritabanına doğru şekilde bağlanmasını ve tabloları oluşturmasını sağlıyor.
* **IdentityDbContext ile Ne Yaptın?**
* Normal bir DbContext sınıfı, senin belirlediğin modeller (Blog, Comment, vb.) için tablolar oluştururken, **IdentityDbContext** sınıfı standart veritabanı tablolarına ek olarak, kullanıcı yönetimi için gerekli olan tabloları da otomatik olarak oluşturuyor. Bu tablolar şunları içerir:
* AspNetUsers: Kullanıcı bilgileri.
* AspNetRoles: Rol bilgileri.
* AspNetUserRoles: Hangi kullanıcının hangi role sahip olduğu bilgisi.
* AspNetUserLogins: Kullanıcıların oturum açma bilgileri (örneğin harici sağlayıcılar).
* AspNetUserClaims: Kullanıcılara ait ek bilgiler.
* Bu sayede, bu kod parçasıyla, kullanıcı kayıt, giriş, şifre yönetimi ve rol tabanlı yetkilendirme gibi karmaşık işlemleri kendi başına yapmana gerek kalmıyor; ASP.NET Core Identity'nin hazır altyapısını kullanabiliyorsun.

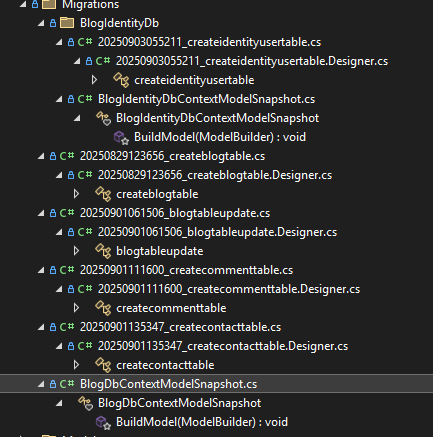
## BlogIdentityRole

* Rol tanımı

## BlogIdentityUser

* Name surname vs tanımlama

# Migrations



### **Veritabanı Geçiş (Migration) Sistemi Detaylı Anlatım**

* Bu sistem, veritabanı yapınızdaki değişiklikleri yönetmenin modern ve hatasız bir yoludur. En basit ifadeyle, veritabanınızın bir nevi **tarihçesini** tutar.

#### **1. Başlangıç Durumu**

* Başlangıçta veritabanınız yoktur. Sadece Blog, Comment, Contact gibi model sınıflarınız vardır. Bu modeller, veritabanınızın nasıl görünmesi gerektiği hakkında EF Core'a bilgi verir.

#### **2. İlk Geçişi Oluşturma (Add-Migration)**

* Projenizde ilk kez Add-Migration komutunu çalıştırdığınızda (örneğin, Add-Migration CreateBlogTables), EF Core şunları yapar:
* **Modelleri tarar**: BlogDbContext içinde tanımladığınız tüm DbSet'leri inceler.
* **Geçiş dosyasını oluşturur**: Modellerinize bakarak, veritabanına bu tabloları (Blogs, Comments, Contacts) nasıl ekleyeceğini belirler. 2025...\_CreateBlogTables.cs gibi yeni bir dosya oluşturur. Bu dosya, Up() ve Down() olmak üzere iki metot içerir:
  + Up(): CreateTable komutlarını içerir. Bu komutlar, veritabanında yeni tabloları oluşturmak için kullanılır.
  + Down(): DropTable komutlarını içerir. Bu komutlar, değişiklikleri geri almak isterseniz tabloları siler.
* **Snapshot oluşturur**: Veritabanınızın o anki durumunun bir anlık görüntüsünü alır. Bu, sonraki geçişleri oluştururken temel olarak kullanılır.

#### **3. Veritabanını Güncelleme (Update-Database)**

* Bu komutu çalıştırdığınızda, EF Core şunları yapar:
* **Bekleyen geçişleri kontrol eder**: Henüz veritabanına uygulanmamış geçiş dosyalarını bulur.
* **Geçişleri uygular**: Up() metodunun içinde yer alan komutları sırasıyla veritabanında çalıştırır. Sonuç olarak, veritabanınızda Blogs, Comments, Contacts gibi tablolar oluşturulur.
* **Geçişi kaydeder**: \_\_EFMigrationsHistory adında özel bir tabloya bu geçişin uygulandığını kaydeder.

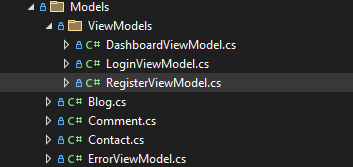
#### **4. Mevcut Yapıya Değişiklik Yapma**

* Diyelim ki Blog modelinize yeni bir Category sütunu eklemek istiyorsunuz.
* **1. Adım (Kod Değişikliği)**: Blog.cs model sınıfına public string Category { get; set; } gibi yeni bir özellik eklersiniz.
* **2. Adım (Yeni Geçiş Oluşturma)**: Add-Migration AddCategoryToBlog gibi yeni bir komut çalıştırırsınız. EF Core şunları yapar:
  + **Snapshot'u inceler**: Mevcut veritabanı yapınızın son durumuna bakar.
  + **Modeli inceler**: Blog modelindeki yeni Category özelliğini fark eder.
  + **Yeni geçiş dosyasını oluşturur**: Veritabanına bu sütunu eklemek için gereken AddColumn komutunu içeren bir geçiş dosyası oluşturur.
* **3. Adım (Veritabanını Güncelleme)**: Update-Database komutunu çalıştırırsınız. EF Core, veritabanında henüz uygulanmamış olan AddCategoryToBlog geçişini bulur ve Blogs tablosuna Category sütununu ekler.

### **Bu Sistemin Sağladığı Avantajlar**

* **Sürüm Kontrolü**: Tıpkı Git gibi, veritabanı şemanızın tüm geçmişini kolayca görebilir ve yönetebilirsiniz.
* **Kolay İşbirliği**: Bir ekip üyesi veritabanında bir değişiklik yaptığında, diğer üyeler Update-Database komutuyla kendi yerel veritabanlarını tek bir komutla en son sürüme güncelleyebilirler.
* **Ortamlar Arası Eşitlik**: Geliştirme, test ve canlı (prod) ortamlarındaki veritabanı yapılarının her zaman aynı olmasını sağlar.
* **Hata Kurtarma**: Bir geçiş işlemi sırasında hata oluşursa, Down() metodu sayesinde önceki stabil duruma kolayca dönebilirsiniz.
* Bu sistem, veritabanı yönetimini manuel SQL komutlarından kurtararak geliştirme sürecini çok daha güvenli ve pratik hale getirir.

# Models



### **Modeller (Models Klasörü)**

* Modeller, uygulamanızın **veritabanı tablolarına karşılık gelen** sınıflardır. Bunlara aynı zamanda **veri modelleri** veya **varlıklar (entities)** da denir.
* **Blog.cs**: Veritabanındaki **Blogs** tablosunu temsil eder. Id, Title, Content, ViewCount gibi özellikler içerir.
* **Comment.cs**: Veritabanındaki **Comments** tablosunu temsil eder. Yorumun içeriği, yazarı ve hangi bloga ait olduğu gibi bilgileri içerir.
* **Contact.cs**: Veritabanındaki **Contacts** tablosunu temsil eder. İletişim formundan gelen ad, e-posta ve mesaj gibi verileri tutar.
* **ErrorViewModel.cs**: Hata sayfalarını yönetmek için kullanılan bir modeldir.
* Bu model sınıfları, Entity Framework Core tarafından veritabanı tablolarına dönüştürülür ve veritabanı işlemlerini nesne yönelimli bir şekilde yapmanızı sağlar.

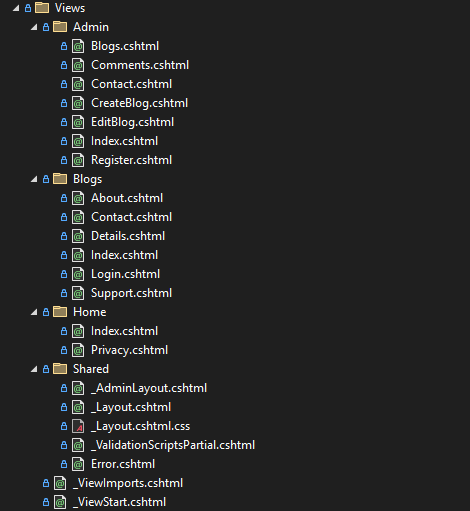
### **Görünüm Modelleri (ViewModels Klasörü)**

* Görünüm modelleri, **belirli bir sayfa (view) için özel olarak oluşturulmuş** sınıflardır. Veritabanı modellerinden farklı olarak, doğrudan veritabanı tablolarına karşılık gelmezler.
* **DashboardViewModel.cs**: Admin paneli ana sayfası (Index.cshtml) için oluşturulmuştur. Bu model, tek bir blog nesnesini değil, toplam blog sayısı, toplam görüntülenme sayısı gibi farklı verileri bir araya getirir.
* **LoginViewModel.cs**: Giriş sayfası için kullanılır. Genellikle Email ve Password gibi özellikleri içerir. Veritabanındaki User modeliyle birebir aynı olmak zorunda değildir, yalnızca giriş formu için gereken bilgileri tutar.
* **RegisterViewModel.cs**: Kayıt sayfası için kullanılır. Name, Surname, Email, Password ve RePassword gibi bilgileri bir arada tutar.

### **Model ve ViewModel Arasındaki Fark**

* **Model**: Veritabanı yapısını yansıtan, doğrudan veritabanı tablolarına karşılık gelen sınıflardır.
* **ViewModel**: Belirli bir sayfanın (view) ihtiyacı olan verileri taşımak için oluşturulan sınıflardır. Bu veriler birden fazla modelden, başka bir kaynaktan veya sadece o sayfaya özel olarak toplanabilir.
* Bu ayrım, uygulamanızın daha düzenli ve güvenli olmasını sağlar. Doğrudan veritabanı modellerini sayfaya göndermek yerine, sadece sayfanın ihtiyaç duyduğu verileri içeren bir ViewModel kullanmak **veri sızıntısını engeller** ve performansı artırır.

# Views



### **MVC Mimarisinde Views Klasörünün Yeri**

* MVC (Model-View-Controller) mimarisinde:
* **Model** (veri)
* **Controller** (iş mantığı)
* **View** (kullanıcı arayüzü)
* olarak üç ana bileşen bulunur. Views klasörü, bu mimarideki **View** bileşenine karşılık gelir. Bir Controller (örneğin BlogsController), iş mantığını tamamladıktan sonra bir View'e veri gönderir. View, bu veriyi alır ve bir web sayfası (.cshtml dosyası) olarak HTML çıktısı üretir.

### **Klasörlerin ve Dosyaların Amacı**

* Views klasörü, projenin düzenini korumak için genellikle denetleyici (controller) isimlerine göre alt klasörlere ayrılır.

#### **Admin Klasörü**

* Bu klasör, **AdminController**'ın kullandığı tüm sayfaları (view) içerir.
* Blogs.cshtml: Admin panelindeki blog listesi sayfası.
* Comments.cshtml: Yorumları yönetme sayfası.
* Contact.cshtml: İletişim mesajlarını yönetme sayfası.
* CreateBlog.cshtml: Yeni blog yazısı oluşturma formu.
* EditBlog.cshtml: Var olan bir blog yazısını düzenleme formu.
* Index.cshtml: Admin paneli ana sayfası (genellikle dashboard).
* Register.cshtml: Admin kullanıcı kaydı sayfası.

#### **Blogs Klasörü**

* Bu klasör, **BlogsController**'ın kullandığı herkese açık (public) sayfaları içerir.
* About.cshtml: Hakkımızda sayfası.
* Contact.cshtml: İletişim formu.
* Details.cshtml: Blog yazısının detay sayfası.
* Index.cshtml: Blog ana sayfası (yayınlanmış blogların listesi).
* Login.cshtml: Kullanıcı giriş sayfası.
* Support.cshtml: Destek sayfası.

#### **Home Klasörü**

* **HomeController** tarafından kullanılan sayfaları içerir.
* Index.cshtml: Uygulamanın ana giriş sayfası.
* Privacy.cshtml: Gizlilik politikası sayfası.

#### **Shared Klasörü**

* Bu klasör, birden fazla sayfada ortak olarak kullanılan dosyaları içerir.
* \_AdminLayout.cshtml ve \_Layout.cshtml: Uygulamanın farklı kısımları için ana sayfa düzenleridir. Header, footer, menü gibi ortak HTML yapılarını içerirler.
* \_Layout.cshtml.css: \_Layout dosyasına ait CSS stil dosyası.
* \_ValidationScriptsPartial.cshtml: Formlarda client-side doğrulama (JavaScript ile) yapmak için gerekli scriptleri içerir.
* Error.cshtml: Uygulamada bir hata oluştuğunda gösterilen sayfa.
* \_ViewImports.cshtml: View'lere otomatik olarak dahil edilecek using direktiflerini içerir.
* \_ViewStart.cshtml: Tüm view'lerin başında çalışır ve hangi \_Layout dosyasının kullanılacağını belirtir.
* Her bir .cshtml dosyası, HTML kodunun yanı sıra C# kodunu da içerebilir. Bu teknolojiye **Razor** adı verilir ve dinamik web sayfaları oluşturmayı mümkün kılar. Örneğin, bir .cshtml dosyasında, bir **@foreach** döngüsü kullanarak bir blog listesindeki her bir blogu ekrana basabilirsin.

# Appsetting.json

* Bu dosya, uygulamanın çalışırken ihtiyaç duyduğu ayarları ve yapılandırma bilgilerini saklamak için kullanılır.
* appsettings.json dosyası, uygulamanın kodundan bağımsız olarak ayarları kolayca değiştirmenizi sağlar. Böylece, veritabanı bağlantı adresi, loglama seviyesi veya API anahtarları gibi bilgileri, kodunuzu yeniden derlemeye gerek kalmadan farklı ortamlara (geliştirme, test, canlı sunucu) göre ayarlayabilirsiniz.

### **İçeriğin Açıklaması**

* **"ConnectionStrings"**: Bu bölüm, uygulamanın veritabanına bağlanmak için kullandığı bağlantı dizelerini (connection strings) içerir.
  + **"DefaultConnection"**: Bu, bağlantı dizesine verdiğiniz bir isimdir. Kodunuzda bu ismi kullanarak veritabanına kolayca bağlanabilirsiniz.
  + **Data Source =NB3\\SQLEXPRESS**: Bağlanılacak veritabanı sunucusunun adını belirtir.
  + **database=blogV1**: Bağlanılacak veritabanının adını belirtir.
  + **Integrated Security=True**: Windows kimlik doğrulamasını kullanarak veritabanına bağlanmayı sağlar. Yani bir kullanıcı adı ve şifreye gerek kalmaz.
  + **TrustServerCertificate=True**: Sunucu sertifikasına güvenileceğini belirtir, bu genellikle yerel geliştirme ortamlarında kullanılır.
* **"Logging"**: Bu bölüm, uygulamanın loglama ayarlarını kontrol eder. Loglama, uygulamanın çalışma zamanındaki olayları (bilgilendirme, uyarı, hata gibi) kaydetme işlemidir.
  + **"LogLevel"**: Logların hangi seviyede kaydedileceğini belirler.
  + **"Default": "Information"**: Uygulamanın genel olarak Information seviyesindeki logları ve daha yüksek seviyeleri (Warning, Error, vb.) kaydetmesini sağlar.
  + **"Microsoft.AspNetCore": "Warning"**: ASP.NET Core altyapısıyla ilgili logların sadece Warning seviyesinde ve daha yüksek seviyelerde kaydedilmesini sağlar. Bu, gereksiz bilgilendirme loglarının önüne geçerek log dosyasını daha temiz tutar.
* **"AllowedHosts"**: Bu bölüm, uygulamanın hangi sunucu adlarına (host) gelen isteklere yanıt vereceğini belirler.
  + **"\*"**: Uygulamanın gelen tüm sunucu adlarına yanıt vermesini sağlar. Bu, geliştirme aşamasında yaygın olarak kullanılır. Canlı ortamda ise güvenlik için belirli bir alan adı veya IP adresi belirtilir.

# Program.cs

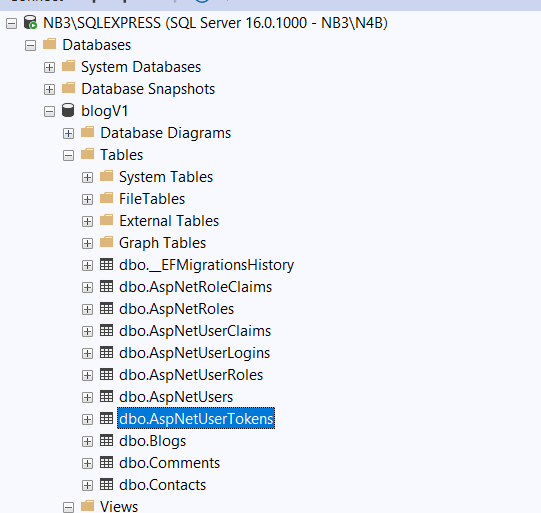
### **İlk Bölüm: Servisleri Tanımlama**

* Bu kısım, uygulamanızın hangi araçları ve hizmetleri kullanacağını söylediğiniz yerdir.
* **var builder = WebApplication.CreateBuilder(args);**
  + Uygulamanı başlatmak için gereken bir "yapıcı" (builder) oluşturur. Bu, tüm ayarların toplandığı merkezdir.
* **builder.Services.AddControllersWithViews();**
  + Uygulamanın **MVC (Model-View-Controller)** mimarisini kullanacağını belirtir. Yani, denetleyiciler ve sayfalar (.cshtml) arasında bir bağ kurulur.
* **builder.Services.AddDbContext<BlogDbContext>();**
  + **BlogDbContext** sınıfını bir hizmet olarak tanımlar. Bu sayede, Controller içinde \_context gibi bir değişkenle veritabanına kolayca erişim sağlarsın.
* **builder.Services.AddDbContext<BlogIdentityDbContext>(options => { ... });**
  + **ASP.NET Core Identity** için özel bir veritabanı bağlantısı kurar.
  + { ... } içindeki kod, **appsettings.json** dosyasından "DefaultConnection" adlı bağlantı dizesini bulur ve bu bilgiyi kullanarak **BlogIdentityDbContext**'i bir SQL Server veritabanına bağlar.
* **builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie(options => { ... });**
  + Uygulamada **kimlik doğrulama** (authentication) servisini başlatır.
  + Oturumları yönetmek için **çerezleri (cookie)** kullanacağını belirtir.
  + **options.LoginPath = "Blogs/Index";**: Eğer bir kullanıcı yetkisi olmayan bir sayfaya erişmeye çalışırsa, onu Blogs/Index adresine yönlendirir.
* **builder.Services.AddIdentity<BlogIdentityUser,BlogIdentityRole>().AddEntityFrameworkStores<BlogIdentityDbContext>().AddDefaultTokenProviders();**
  + **Identity** sisteminin kendisini yapılandırır.
  + Hangi kullanıcı ve rol sınıflarını (BlogIdentityUser, BlogIdentityRole) kullanacağını belirtir.
  + Kullanıcı ve rol verilerini yönetmek için **Entity Framework Core**'u kullanacağını söyler.

### **İkinci Bölüm: Uygulamanın Davranışlarını Tanımlama**

* Bu kısım, bir web isteği geldiğinde uygulamanın nasıl davranacağını adım adım belirler.
* **var app = builder.Build();**
  + Tanımlanan tüm servisler ve ayarlar ile birlikte çalışan bir uygulama nesnesi (app) oluşturur.
* **if (!app.Environment.IsDevelopment()) { ... }**
  + Eğer uygulama "geliştirme ortamında" değilse (yani canlı sunucudaysa), özel ayarların devreye girmesini sağlar.
  + **app.UseExceptionHandler("/Home/Error");**: Canlı ortamda bir hata olursa, kullanıcıya genel bir hata sayfası gösterir.
  + **app.UseHsts();**: Tarayıcıların siteye her zaman güvenli (HTTPS) bağlantı üzerinden erişmesini zorunlu kılar.
* **app.UseHttpsRedirection();**
  + Gelen tüm HTTP isteklerini otomatik olarak güvenli HTTPS isteklerine dönüştürür.
* **app.UseStaticFiles();**
  + CSS, JavaScript, resimler gibi statik dosyaların tarayıcıya sunulmasını sağlar.
* **app.UseRouting();**
  + Gelen isteğin URL adresine bakarak, hangi Controller metodunun çalışacağını belirlemeye hazırlanır.
* **app.UseAuthentication();**
  + Kullanıcının kimliğini doğrular. Örneğin, çerezlere bakarak kullanıcının kim olduğunu anlar.
* **app.UseAuthorization();**
  + Kullanıcının belirli bir sayfaya veya fonksiyona erişim yetkisinin olup olmadığını kontrol eder.
* **app.MapControllerRoute(...)**
  + URL adreslerinin nasıl okunacağını belirleyen bir rota (route) kuralı tanımlar.
  + **pattern: "{controller=Blogs}/{action=Index}/{id?}"**: Varsayılan URL yapısının site.com/Blogs/Index gibi olacağını belirtir. Eğer URL'de bir şey yazılmazsa, otomatik olarak Blogs denetleyicisindeki Index metodunu çalıştırır.
* **app.Run();**
  + Uygulamayı başlatır ve gelen web isteklerini dinlemeye başlar. Uygulamanın çalışması bu komutla başlar.

# Sql ile Bağlantı



* Ekran görüntüsünde gördüğün veritabanı tabloları, senin C# kodundaki **model sınıfların** ve **Entity Framework Core (EF Core) Migrations** sistemi aracılığıyla programına bağlanır ve oluşturulur.
* Bu tabloların kendiliğinden oluşmasını sağlayan süreç şu şekildedir:

### **1. Model Sınıflarını Yazmak**

* İlk olarak, programında veritabanındaki tabloları temsil eden C# sınıflarını yazarsın. Örneğin:
* **Blogs**, **Comments** ve **Contacts** tabloları için sırasıyla Blog.cs, Comment.cs ve Contact.cs gibi model sınıfları oluşturursun. Bu sınıfların içindeki özellikler (properties), tablolardaki sütunlara karşılık gelir.

### **2. Veritabanı Bağlamını (DbContext) Tanımlamak**

* Daha sonra, bu model sınıflarını bir araya getiren bir veritabanı bağlam sınıfı (DbContext) oluşturursun. Örneğin:
* C#
* public class BlogDbContext : DbContext  
  {  
   // Blogs tablosunu Blog modeliyle eşleştirir  
   public DbSet<Blog> Blogs { get; set; }  
   // Comments tablosunu Comment modeliyle eşleştirir  
   public DbSet<Comment> Comments { get; set; }  
   // Contacts tablosunu Contact modeliyle eşleştirir  
   public DbSet<Contact> Contacts { get; set; }  
  }
* Bu, Entity Framework Core'a veritabanında hangi tabloların olması gerektiğini söyler.

### **3. IdentityDbContext'i Ayarlamak**

* ASP.NET Core Identity için de benzer bir süreç işler:
* **AspNetUsers**, **AspNetRoles** ve diğer AspNet\* tabloları, senin BlogIdentityUser ve BlogIdentityRole gibi sınıflarını kullanarak **IdentityDbContext** sınıfı tarafından otomatik olarak tanımlanır. Bu, kullanıcı yönetimi için gerekli tüm tabloları içerir.

### **4. Migrations Oluşturmak**

* Kodunu yazdıktan sonra, Visual Studio'daki Paket Yöneticisi Konsolu'nu (Package Manager Console) kullanarak migrasyon (migration) komutlarını çalıştırırsın.
* Add-Migration [GeçişAdı]
  + Bu komut, EF Core'un kodundaki model sınıflarını ve veritabanının mevcut durumunu karşılaştırmasını sağlar.
  + Farklılıkları (Blog tablosunu oluşturmak gibi) algılar ve bu değişiklikleri gerçekleştirecek C# kodunu içeren migrasyon dosyalarını **Migrations** klasöründe otomatik olarak oluşturur.

### **5. Veritabanını Güncellemek**

* Son olarak, oluşturulan migrasyon dosyalarını veritabanına uygulamak için şu komutu çalıştırırsın:
* Update-Database
  + Bu komut, EF Core'a Migrations klasöründeki henüz uygulanmamış geçiş dosyalarını çalıştırmasını söyler.
  + EF Core, bu dosyaların içindeki kodları (örneğin CreateTable("Blogs", ...)) bir **SQL** sorgusuna çevirir ve bu sorguyu veritabanında çalıştırır.
* Bu adımlar tamamlandığında, senin model sınıflarına karşılık gelen tablolar (Blogs, Comments, vb.) ve Identity için gerekli tablolar (AspNetUsers, AspNetRoles, vb.) veritabanında oluşturulmuş olur. Artık kodunla bu tablolara erişebilir, veri ekleyip çıkarabilir veya güncelleyebilirsin.

# ÖZET

* Bizim elimizde temelinde 3 bileşen var model view controller.
* Once modelde orda hangı degıskenlerı kullanıcaksak onları tanımlarız. Bunlar aslında tabloların bır gorunumudur.
* Sonrasında biz backend tarafını yaparız controllerda yapılır. Blog ıcın admın ve blog diye iki ayrı controller yapılır
* Sonrasında biz bu controllerda yazdıklarımızı model yapısını kullanarak view üzerinden gorunumunu verırız.
* Goruntu kısmında her bır methodun ayrı bır goruntusu ve klasık blog ve admın sayfası ıcın goruntuler vardır. Bu goruntulerı ınternetten şemalar indirerek sağlayıp projeye gereken yerlerini dahil ederiz.