MVC

MVC bir frameworktür.

Cross platform için Core eklendi. .net 6 dan sonra Core kavramı .Net ten kaldırıldı. ASP.Net içinde hala var.

1. **ASP.NET Core Empty proje aç**
2. **Controllerlar ve geri dönülecek View lar kullanacağını program.cs e ekle:**

builder.Services.AddControllerWithViews();

AddControllersWithViews, kontrolörler ve görünümler ile MVC uygulamaları oluşturmak için gerekli bileşenleri ekler. Bu hizmet, kontrolör ve eylem kullanımını sağlar ve görünümlerin oluşturulmasına izin verir.

AddMvc ise, MVC çerçevesinin tamamını uygulamaya ekler. Bu hizmet, kontrolör ve görünümlerin yanı sıra API kontrolörleri, model bağlama, yönlendirme ve diğer gelişmiş özellikleri de ekler.

AddControllersWithViews kontrolör ve görünümler ile sınırlı MVC uygulamaları oluşturmak için kullanılırken, AddMvc daha kapsamlı MVC uygulamaları oluşturmak için kullanılır.

app.MapControllerRoute(

name:"Default" → bu siteye özel bir istek atılmadığı zaman (admin paneli vb.),

pattern: "{Controller=Home}/{Aciton=Index}/{id?}" → Nasıl istek atacağını gösteren kısım. Oto anasayfa belirle. id kısmı optional

);

- app.Run(); → hep en sonda
- builder. İle başlayan kodlar hep → var.app = builder.Build(); komutunun üstünde

3. **Controllers Klasörü ekle**

Anasayfa, hakkında gibi temel işlemler için Home/MainController. Onun dışında modellerimiz için C R U D (Create, Read, Update, Delete) işlemlerini yapabileceğimiz controllerlar.

- Controllerlar aslında classtır. Controller yapan şey .Net frameworkünün içinde :Controller abstract classından miras alır.
- Controller içindeki metodun tipi IActionResult → Interface
  Controller içindeki IActionResult dönen yapılara → ACTION

Solid prensipleri: Alt katmandaki yapılar üst katmandaki yapılarla yer değiştirebilir.

- Controllerdaki actiondan veri çekeceksek
  Public IActionResult Add(TaskAddViewModel formData)
  Artık bu view da girilen girdiler formData da tutulur.

- Controller içine modellerimizi kulanabilmek için statik bir liste oluşturabiliriz. Burda newleme yaparız.

```
Static List<Task> _tasks = new List<Task>(){
Newleme işlemleri
}
```

- Sadece IsDeleted false elemanları görmek isteriz. O yüzden action içinde

```
var ViewModel = _task.Where(x=> x.IsDeleted == false).Select(x => new TaskListViewModel{
Id = x.id, // Böylece yeni taskListViewModelime propları eşliyorum
}.ToList(); → Tüm Linq lar bittikten sonra listeye çevir
return View(viewModel); → ilgili view e gönderdik
```

- View içinde listenin elemanlarını kullanmak için foreach yazarız.
  var task in Model

4. **Views ekle**
   Action adı ile View adı farklı olursa. Return View Action ile aynı adda bir View döndürmeye çalışır. Böyle durumlarda return.View("ViewAdı"); içerisinde belirtilmeli.

- asp-route-id ve benzeri tag helper'lar, URL'e bir query string ya da route parametresi ekler. Özellikle bir id parametresini belirli bir URL'ye geçirmek için kullanılır.
  Bu tag helper, action ya da controller bilgileriyle kullanılabilir ya da sadece bir yönlendirme parametresi ile URL oluşturabilir.
  Örn : asp-route-id=”@task.Id”

5. (ilkel yöntem) Boostrap ekleme

View içerisine html ana iskeleti açarak head tagleri arasına Boostrap web sitesinden alınan linki ekleme. Daha sonra boostrapten istenilen yapılar çekilip entegre edilebilir.

- Webten çekmek istemezsek statik dosya için --> program.cs e

app.UseStaticFiles();

- Folder ekle --> wwwroot dosyası ekle içine bir folder daha --> lib --> boostrap.css .txt dosyası

Bu dosyanın içine bootstrap sitesindeki adresi açıp kodlamaları manuel olarak kopyalayıp yapıştırıyoruz.

- _Layout a gidip head kısmına link href olarak ekliyoruz. =”lib/boostrap.css”

Bu yöntemle navbar eklemek için. Shared klasörüne View ekliyorum adlandırması → _navPartial navbar kodlamalarını bunun içerisine yapıyoruz.

_Layout da containerın üst kısmına navbarı ekliyoruz.

<partial name=”_navPartial>

6. Sitede Çerçeve Kullanma

- Shared adlı bir klasör aç. İçerisine View aç → _Layout //Ortak olarak kullanılan yapılar genellikle “_” ile başlar.
- _Layaout içine html iskeleti koydum ve head kısmına Bootstrap linki ekledim.
- Body kısmına @RenderBody() **(Noktalı virgülsüz!!)**

Razor View içine (.cshtml) hem c# hem html yazılabilir.

@(razor) koyduktan sonra yazılan yazı c# dilinde algılanır.

@{ Bu alan içine c# yazılabilir anlamına gelir

}

- **@{**
  Layout = "_Layout"; → string olarak ister adı
  }
  Bu kodu eklediğim view çerçevemin formatında olur.

7. Layoutu bütün Viewlerde kullanmak için

- Views klasörünün içine → **_ViewStart** adında bir View açtım
  Razorda yorum satırı @* ..... *@

- **@{**
  Layout = "_Layout";
  }

Bu kodu _ViewStart a eklersen. Bütün viewların başına yazılmış gibi kabul edilir.

8. Tag Helpers ın Projeye eklenmesi

- _ViewImports aç

Ortak bir yapı ama Shared içine değil Views içine view olarak açıyoruz.

- İçine tag helpers eklemelerini yapıyoruz (asp tag helpers web site Microsoft) @add
- asp-controller="Controller Adı" asp-action="Action Adı" bu tag helperlar yardımıcı ile tasarıma eklenen buton link vsler için tetikleme yapabiliriz.

Viewlara içinde bulundukları actiondan veriler gönderilir.

Veri Taşıma Yöntemleri: 1-ViewBag 2-TempData 3-ViewData

View ı besleyen action içinde → ViewBag.Kisi = "Ajda Pekkan";

View içinde →@ViewBag.Kisi yazınca çekmiş oluruz

ViewBag kullanmadan;

Örn: int yas = 20; //action içinde

return View(yas); → parametre olarak gönderdik

bunu viewda alabilmek için sayfanın başına gelip→ @model int diyorum böylece sana bir integer veri göndericem demiş oluyoruz.

Gelen veri Model → büyük 'M'

Örn: Viewda veriyi kullanırken Yaş = @Model

Tip belirten model küçük 'm' / en tepede yazılır.

9. **Classlarımızı içerecek Models klasörü**
   - Açtığımız _ViewImports klasörüne

@using ProjeAdı.Models kodunu ekliyoruz böylece her bir viewda artık models klasörümüzdeki classlara erişimimiz sağlanmış oluyor.

- Classımı tanımladıktan sonra. Action içinde newleme yapabilirim. Newlediğim değişkeni view da kullanabilmem için return içinde parametre olarak gönderiyorum.
- Propertyleri tek tek yazdırmak için @Model.*FirstName*

Örnek bir CompletedDate propu tanımlarsak default bir değer alır. Bu değeri görmemek için nullable yapabiliriz typının yanına ? koyarak → DateTime?

Hard Delete

Bir kaydın veritabanından tamamen silinmesi işlemidir. Kaydedilmiş veri, sorgulamalarda artık hiçbir şekilde görünmez ve geri getirilemez.

Soft Delete

Veri silinmez, ancak belirli bir bayrakla (genellikle is_deleted gibi bir alanla) işaretlenerek artık aktif olarak kullanılmaz. Veritabanında kalmaya devam eder, fakat sistemde görünmez.

Kullanmak için propa boolean bir IsDeleted propu ekleriz.

Veri Anotasyonları (Data Annotations)

Veri anotasyonları, sınıf ve property'ler üzerinde doğrudan tanımlanan ve doğrulama kuralları ekleyen özniteliklerdir. Bunlar, modelin kullanıcı girdilerine karşı ne tür doğrulama kuralları

uygulayacağını belirler ve genellikle form doğrulaması ve veri tutarlılığı sağlamak için kullanılır.

Genel Kullanılan Veri Anotasyonları

1. **[Required]**
   - Belirli bir property'sinin değerinin girilmesini zorunlu kılar.
2. **[StringLength]**
   - Bir string property'sinin minimum ve maksimum uzunluk sınırlarını belirler.
3. **[Range]**
   - Bir property'sinin belirli bir değer aralığında olup olmadığını kontrol eder.
4. **[RegularExpression]**
   - Property'nin belirli bir regex (düzenli ifade) desenine uymasını sağlar.
5. **[EmailAddress]**
   - Property'nin geçerli bir e-posta adresi formatında olup olmadığını doğrular.
6. **[Phone]**
   - Property'nin geçerli bir telefon numarası formatında olup olmadığını doğrular.
7. **[Compare]**
   - İki property'sinin eşit olup olmadığını doğrular, genellikle şifre doğrulaması için kullanılır.
8. **[CreditCard]**
   - Property'nin geçerli bir kredi kartı numarası formatında olup olmadığını doğrular.
9. **[Url]**
   - Property'nin geçerli bir URL formatında olup olmadığını doğrular.
10. **[Range]**
    - Bir property'sinin belirli bir değer aralığında olup olmadığını kontrol eder.

Bunların otomatik hata mesajları vardır. Bunu değişmek için örn: [Required (ErrorMessage = “Göstermek istediğiniz hata mesajı”)]

Bunları kullanarak Model Validation yapabiliriz.

Model Validation, bir uygulamada kullanıcıdan veya başka kaynaklardan alınan verilerin doğruluğunu ve geçerliliğini kontrol

etmek için kullanılan bir tekniktir. Özellikle ASP.NET Core gibi web geliştirme frameworklerinde, model doğrulama, kullanıcı girdilerini veya veri modelini belirli kurallara göre kontrol etmek için uygulanır.

View da <span asp-validation-for”Title” class =form-control” type=”text”> kodlaması ile ilgili alanın kontrolünü sağlarız. Action içinde ise bir if yapılandırması ile:

```
if(!ModelState.IsValid){
Return   View(formData);
}
```

böylece geçerli olup olmadığı kontrol ederiz. Geçersiz ise tekrar Viewa döneriz.

View a kullanılmayacak veri gönderilmemeli. Bunun için ViewModels kavramı.

10. **ViewModels klasörü aç**
    - İçerisine class açıyoruz adlandırma örneği: StudentListViewModel
    - ViewModelin içine taşımak istediğimiz propları yazdık.
    - Bağlantılı action içine newleme yaptık. Returnde gönderdik yine ve tekrar view içinde @model ile tipini belirttik
    - Bütün viewlarda View Modellerimizi kullanabilmek için _ViewImports dosyamıza → @using ProjeAdı.ViewModels kodumuzu da ekledik.
11. **Viewdan veri çekmek**
    - İlgili controllera geldim. Şu an verilerimi statik bir şekilde çekicem o yüzden bir list oluşturdum.
    - static list<ClassımınAdı> userList = new List <User>{
      new User { proplarını verdim},
      };
    - Method Overloading yaparak actionımdan bir tane daha yazıyorum ama içine bir return eklemiyorum. Bu http post işlemlerimi gerçekleştirecek.

Bir tanesi Forma Gitme işlemi için linkten tetiklenen ve form sayfasını açan action (HTTPGET)
Diğeri formda gönder butonuna tıklanınca verileri yakalayıp işlemi devam ettiren action (HTTPPOST)

Link üzerinden atılan istek yani tetiklenen action → HTTPGET

Form üzerinden atılan istek yani tetiklenen action → HTTPPOST

- İlgili sayfanın başına kullanacağımız modeli belirtiyoruz örn:
  @model User
- Viewda kullanıcıdan veri alırken örneğin bir form kullanıyorsak doldurulan verinin ne için olduğunu belirtmek için tag helpers kullanırız. İnput içinde asp-for="modelin içerisindeki propun adı"

Butonun <form> içerisinde olduğuna ve type ının submit olduğuna emin ol!!

- Formun içinde nereye gideceğimizi tanımlıyoruz.
  <form method="get" asp-controller="User" asp-action="SignUp">
  Method = get olursa gönderilen yapılar url de gözükür. O yüzden post kullanmak daha mantıklı.
- Viewdan gelen veriyi methodda okuyabilmek için. Httppost olan actiona parametre veririz.
  Örn: public IActionResult SignUp(User formData)
  User tipinde formData değişkeni.
- Karışıklılık olmaması için actionların üzerine [HttpGet] [HttpPost] olduklarını belirt.

Eğer gelen formData verisinin tipi ile listenin veya data base in kabul ettiği veri tipi uyuşmuyorsa, istenilen tipte yapı oluşturup verileri aktarıp öyle listeye veya db e eklenir.

- Artık listeme aldığım verileri ekliyorum.
  user.List.Add(formData);

Bir action sonrası farklı bir sayfaya yönlendirme yapmak için:

return RedirectToAction("ActionAdı", "ControllerAdı");

12. Entity Klasörü

Bir model, veritabanındaki bir tabloyu temsil etmek zorunda değildir. UI katmanına veri taşımak, kullanıcıdan gelen veriyi almak veya farklı veri kaynaklarını birleştirerek iş mantığını yansıtmak için kullanılır.

Model, genellikle birden fazla entity'den gelen verileri birleştirip işleyebilir veya sadece kullanıcı arayüzüne uygun veri taşımak için kullanılan yapılar olabilir. Ayrıca iş kuralları, doğrulama gibi mantıklar bu yapıda yer alabilir.
Entity'nin ana görevi, veritabanındaki bir varlığı temsil etmek ve veritabanıyla etkileşimi sağlamaktır. Entity, veritabanına kayıtlı verilerin taşındığı ve yönetildiği bir yapı olarak işlev görür.

- Projemizde direkt bir "Entities" adlı folder açıyoruz.
- İçerisine sınıf ekleyip adlandırıyoruz → TaskEntity
- İçerisine proplarımızı ekliyoruz.

Single Responsibility Principle (SRP)'nin Temel Fikri
SRP'ye göre, bir sınıfın yalnızca tek bir iş yapması ve bu işin tam olarak ne olduğu açıkça belirlenmelidir. Yani, bir sınıfın değişmesi için yalnızca tek bir sebep olmalıdır. Eğer bir sınıf birden fazla sorumluluğa sahip olursa , o sınıf birden fazla nedenden dolayı değişiklik gerektirebilir ve bu da karmaşıklığı artırır.

Turner If Kullanımı

condition ? trueExpression : falseExpression;

Toggle kavramı, yazılım ve kullanıcı arayüzü tasarımında, bir şeyin iki durum arasında geçiş yapmasını ifade eder. En yaygın kullanımı, bir özellik veya ayarın **açık** (on) ve **kapalı** (off) durumları arasında geçiş yapmasını sağlayan bir mantıktır.