ASP .net core

MVC

(Model, View, Controller)

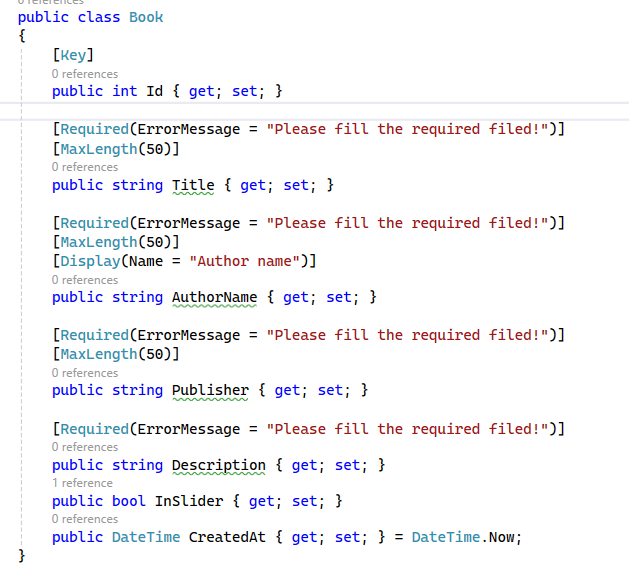
**بخش اول ( اتصال به پایگاه داده )**

برای ساخت یک پروژه متصل به پایگاه داده مراحل زیر را باید طی کنیم:

1. نرم افزار پایگاه داده مورد نظر که در اینجا SQL Server است را باز کرده و به پایگاه داده متصل شوید.شناسه اتصال را کپی کنید تا در Connection String آن را استفاده کنید.
2. مدل مورد نظر را طراحی کنید:

برای ایجاد مدل کافی است که یک کلاس با نام مدل مورد نظر مثلا Book ایجاد کنید و در آن property هایی مثل نام و آی دی ایجاد کنید.

خصوصیات میتوانند دارای محدودیت ها و مشخصاتی باشند. برای مثال خصوصیتی کلیدی ای مثل code یا id باید به عنوان کلید اصلی مطرح شوند. پس به صورت زیر آن ها را تعریف می کنیم:



* عبارت [key] قبل از خصوصیت به معنای تعیین کلید اصلی است.
* عبارت [required] نیز میتواند یه رشته پیام را دز صورت خالی بودن فیلد با عنوان پیغام خطا ارسال کند.
* همچنین می توان توسط [MaxLength] طول دریافت رشته را مشخص کرد.

1. پس از ساخت مدل پکیج های زیر را نصب کنید:

Microsoft entity framework core

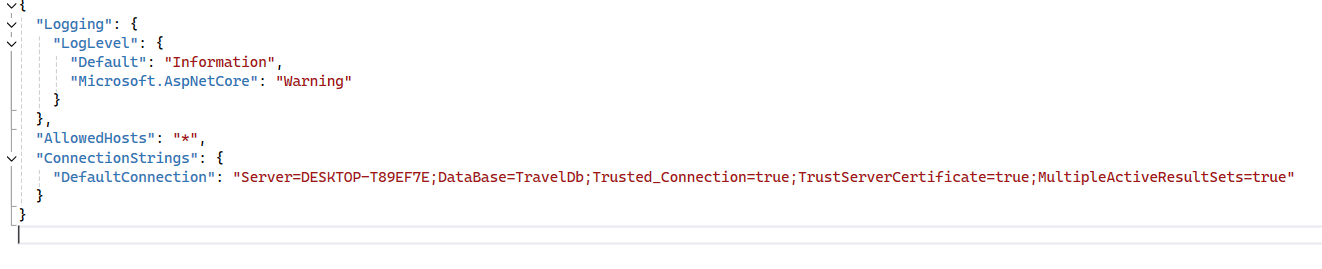
Microsoft entity framework core design

Microsoft entity framework core SQL server

Microsoft entity framework core Tool

پس از نصب پکیج ها باید در Dependencies و در پوشه Package پکیج های نصب شده را ببینید.

1. سپس باید به فایل appsetting.json رفته و Connection String را بنویسید:

جلوی نام سرور شناسه اتصالی که کپی کردیم را مینویسیم. و جلوی نام دیتابیس نام جدولی که میخواهیم در پایگاه داده ساخته شود آورده میشود.

“ConnectionStrings”: {

“DefaultConnection”: “Server=DESKTOP-T89EF7E;DataBase=TravelDb;Trusted\_Connection=true;TrustServerCertificate=true;MultipleActiveResultSets=true” {

1. پوشه ای با نام Data ساخته و در آن کلاسی به نام ApplicationDbContext بسازید.

این کلاس باید از DbContext ارث بری کند. Using را در فضای نامی بنویسید تا کلاس بتواند از پکیج دانلود شده استفاده کند.

Using Microsoft.EntityFrameworkCore;

سپس CTOR از کلاس می سازیم:

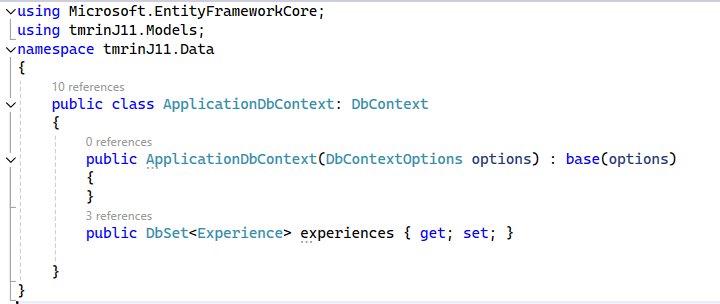
public ApplicationDbContext (DbContextOptions options) : base(options)

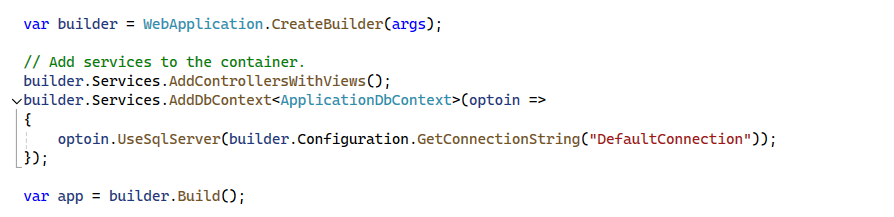
{

}

و یک DbSet از نوع مدلمان می سازیم.

public DbSet<Book> book { get; set; }



1. در این مرحله باید DbContext را در میدل ور (program.cs) اضافه کنیم.

builder.Services.AddDbContext<ApplicationDbContext>(optoin =>

{

optoin.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));

});

1. در مرحله آخر جدول ها را میسازیم. برای ساخت جدول از ممنوی Tools روی console کلیک کرده و در کنسول دو دستور زیر را بنویسید

Add-migration table name

Update-database

در صورت مشاهده خطای قرمز اتصال به پایگاه داده به درستی انجام نگرفته.

1. سپس میتوانید ActionResult هایی مثل index ,Delete, Create و غیره بنویسید و به اکشن ها view بدهید.

**بخش دوم (CRUD )**

به چهار عمل Create, Read , Update , Delete به اختصار CRUD گفته می شود.

برای ایجاد این تغییرات باید برای هر عمل یک اکشن متد در کنترلر مربوطه بنویسید.

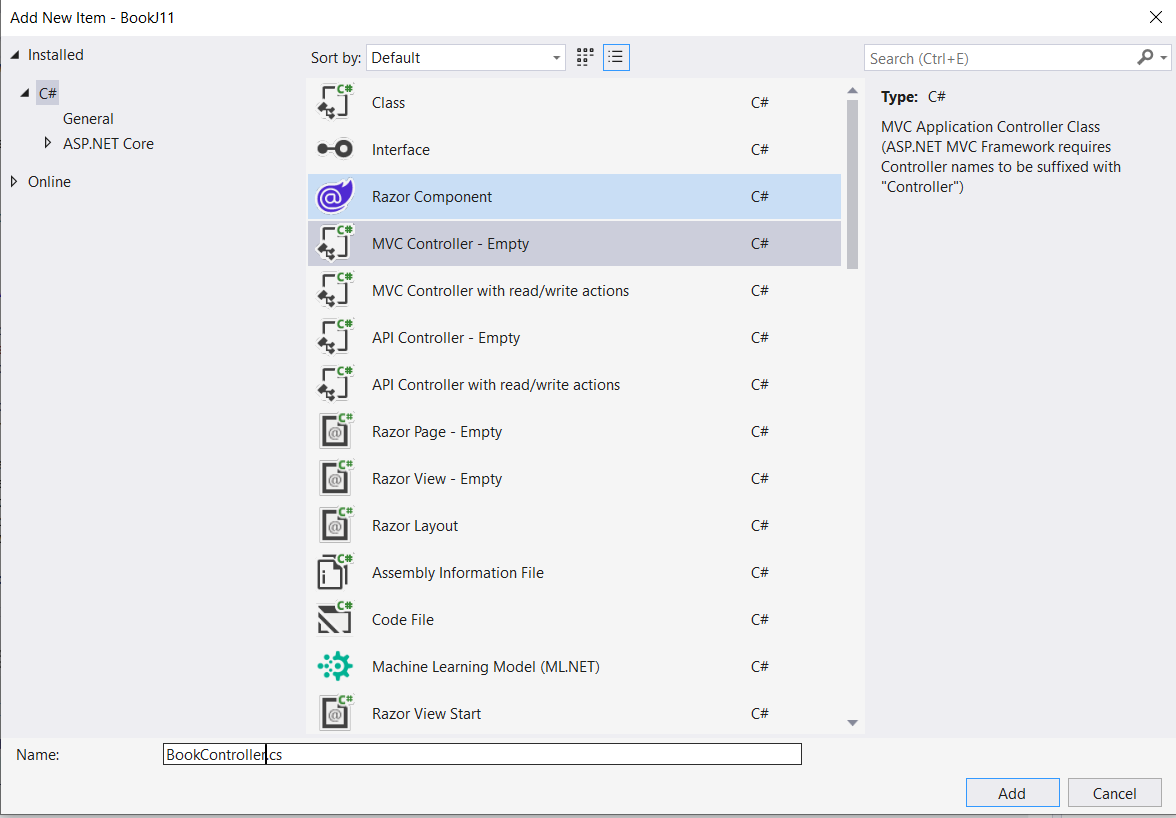
کنترلر مکانی است که دستورات و توابع نوشته میشود.

برای ساخت یک سایت کتاب و کتاب خوانی ما به سه کنترلر نیاز داریم:

HomeController و DashboardController برای نمایش اطلاعات و BookController برای حذف و اضافه کردن به لیست کتاب ها.

برای ساخت کنترلر بر روی پوشه controllers کلیک راست کرده و controller را Add کنید.

💡) به یاد داشته باشید که انتهای نام کنترلر ها عبارت Controller قرار میگیرد.



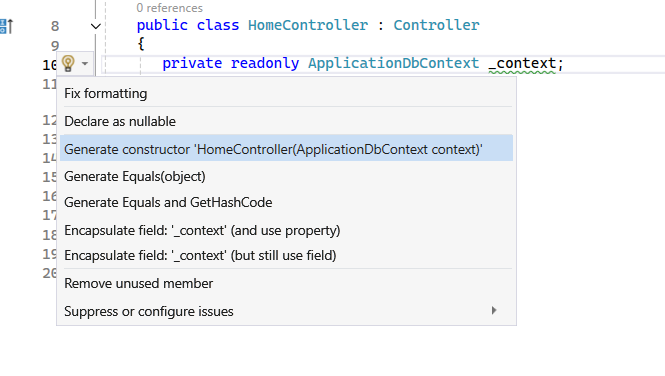
برای اینکه عملی بر روی اطلاعات اتفاق بی افتد، در امرحله اول باید به پایگاه داده متصل شویم و در مرحله بعدی برای استفاده از کلاس ApplicationDbContext در کنترلر ها ما نیاز به ساخت نمونه و دسترسی به خصوصیات این کلاس داریم.

1. در هر کدام از کنترلر های ساخته شده یک نمونه با نام \_context و یک CTOR برای ApplicationDbContext می سازیم.

private readonly ApplicationDbContext \_context;

💡) خط تیره قبل از context نشان از این است که متغیری محلی است. همچنین این متغیر به دلیل حفظ امنیت باید از نوع private و readonly باشد.

1. پس از ساخت نمونه روی انتهای کد قبلی ایست کرده و Ctrl + . را فشار میدهیم تا برای کلاس یک CTOR ایجاد کنیم.



1. پس از انتخاب گزینه مورد نظر ، کد زیر به دستورات اضافه میشود.

public HomeController(ApplicationDbContext context)

{

\_context = context; }

این مراحل را برای هر سه کنترلر طی کنید.

حالا میتوانیم چهار عمل CRUD را بر روی جداول پیاده سازی کنیم. برای ایجاد اکشن های حذف و اضافه به BookController بروید:

💡) از آنجایی که نمایش اطلاعات به عهده کنترلر های دیگر است، اکشن Index را از BookController پاک میکنیم.

**اکشن Create :**

این متد در مرحله اول یک صفحه برای اضافه کردن به لیست کتاب ها به ما نشان میدهد و در مرحله دوم اطلاعات را توسط یک دکمه submit در پایگاه داده ذخیره میکند. که هر کدام این اعمال یک متد مجزا اما نام مشترک نیاز دارند:

[HttpGet] //Get: book/create

public IActionResult Create()

{

return View();

}

متد اول ما را به صفحه Create هدایت میکند. این متد هیچ ورودی ای ندارد و در خروجی یک View برمیگرداند.

* با کلیک راست بر روی Create و انتخاب Add View میتوان یک صفحه .cshtml ساخت که در زمان اجرای متد نمایش داده میشود.

اکشن متد دوم یک متد است با همان نام Create اما در ورودی Book را دریافت میکند تا اطلاعات جدید را به جدول اضافه کند.

💡) [HttpGet] و [HttpPost] مجوزی است که به متد اجازه دریافت و ارسال اطلاعات را میدهد. متد اول از جنس Get و متد دوم از جنس Post است.

💡) در صورتی که مقداری قبل از متد ذکر نشود به طور پیش فرض متد از نوع [HttpGet] خواهد بود.

public IActionResult Create(Book book)

{

if (ModelState.IsValid) {

\_context.Books.Add(book);

\_context.SaveChanges();

return RedirectToAction("Index","Dashboard");

}

return View();

}

در مدل برای خصوصیات کتاب محدودیت هایی قرار دادیم که با شرطی درستی آن را چک می کنیم و در صورت درست بودن مقادیر ورودی توسط متد Add به نمونه جدول مقدار ورودی را اضافه میکنیم و سپس توسط توسط تابعی آن را در جدول ذخیره میکنیم.

ما یک کنترلر به نام Dashboard داریم که دارای متدی به نام Index برایی نمایش اطلاعات است. میخواهیم وقتی به لیست اضافه شد به صفحه داشبورد برویم. این متد در ادامه نوشته می شود.

💡) SaveChanges معادل Add-migration است و همان کار را انجام می دهد.

**اکشن Update :**

برای ایجاد تغییرات روی اطلاعات نیز به دو متد هم نام ولی متفاوت نیاز است. در متد اول که از جنس Get است id دریافت کرده و به دنبال آن در جدول میگردیم.

[HttpGet] // GET: book/edit

public IActionResult Edit(int id)

{

var book = \_context.Books.Find(id);

return View(book);

}

متغیری با نام book میسازیم و در نمونه ای که از جدول ساختیم به دنبال id که در ورودی گرفته بودیم میگردیم. نتیجه این جست و جو در متغیر ساخته شده ریخته می شود.

حالا ما متغیر را به صفحه پاس میدهیم.

💡) در ادامه برای Edit یک view هم میسازیم.

عمل اصلی آپدیت در متد دوم انجام میگیرد. متد دوم خاصیت ارسال اطلاعات یا همان post را در خود دارد:

[HttpPost] // POST: book/edit

public IActionResult Edit(Book book)

{

if (ModelState.IsValid)

{

\_context.Books.Update(book);

\_context.SaveChanges();

return RedirectToAction("Index", "Dashboard");

}

return View();

}

متد در ورودی کلاس Book را دریافت میکند زیرا اطلاعات دریافتی به صورت کلاس است. سپس با شرط پر بودن و محدودیت های تکست باکس ها چک میشود.

در صورت درست بودن جدول کتاب ها را آپدیت میکنیم. این تابع به عنوان ورودی اطلاعات دریافت شده را میگیرد.

سپس باید تغییرات در پایگاه داده ذخیره شود و متد Index در کنترلر Dashboard به نمایش در آید.

**اکشن Delete :**

در این اکشن ما فقط نیاز به حذف یک آیتم داریم. در اکشن های قبلی بعد از اعمال تغییرات آن ها را به نمایش نیز در می آوردیم.

کار این اکشن فقط ارسال اطلاعات به پایگاه داده است پس فقط از post استفاده می شود.

[HttpPost] // POST: book/delete

public IActionResult Delete(int id)

{

var bookToDelete = \_context.Books.Find(id);

if (bookToDelete != null)

{

\_context.Books.Remove(bookToDelete);

\_context.SaveChanges();

}

return RedirectToAction("Index", "Dashboard");

}

ابتدا id کتابی که میخواهیم حذف شود را دریافت میکنیم. سپس در جدول کتاب ها به دنبال آن گشته و آن را در متغیری ذخیره میکنیم.

اگر مقدار پیدا شود شرط bookToDelete != null درست است و میتوانیم با تابع Remove متغیر حاوی اطلاعات را از جدول حذف کنیم.

سپس باید تغییرات را در پایگاه داده ذخیره کنیم و توسط دستور RedirectToAction به اکشن Index در کنترلر Dashboard برویم.

**بخش سوم (Layout ها و سکشن بندی)**

برای داشتن رابط کاربری بهتر ما نیاز به فایل هاییcss و js و غیره داریم. همه فایل های خارجی در پوشه [www.root](http://www.root) قرار میگیرند. صفحات وب میتوانند دارای قالب هایی باشند. به عنوان مثال به صورت پیش فرض قالب \_layout بر روی صفحاتی که قالبی ندارند اعمل می شود. فایل این قالب در پوشه shared قرار دارد و هر فایلی که در این پوشه قرار گیرد در کل برنامه قابل استفاده است. در این پوشه فایل cshtml دیگری نیز وجود دارد به نام \_ViewStart که مشخص میکند صفحاتی که قالبی ندارند از چه قالبی پیروی کنند.

* میخواهیم برای اکشن ها view بسازیم. در واقع میخواهیم قالبی برای این پروژه طراحی کنیم:

روی پوشه shared کلیک راست کرده و یک razor view empty با نام \_MainLayout ایجاد کنید.

💡) پیش فرض را در \_ViewStart به \_MainLayout تغییر دهید.



فایل index.html که جزو فایل های کمکی این جلسه است را باز کنید و محتویات آن را در \_MainLayout بریزید.

💡) برای آدرس دهی به ریشه اصلی یا همان root به جای . از ~ استفاده کنید.

* فایل های asset و css و js و style را از پوشه کمکی در پوشه روت کپی کنید. مسیر فایل روت به صورت زیر است:

C:\Users\ACER\source\repos\BookJ11\BookJ11\wwwroot

💡) فایل style را نیز در پوشه css قرار دهید.

در قالب ها سر صفحه ها ، سربرگ ها و فوتر ها ثابت است. وسط صفحه است که با توجه به دستورات تغییر خواهد کرد.

میخواهیم اکشن ها را در وسط صفحه به نمایش در بیاوریم. برای انجام این کار به دستور RenderBody() نیاز است. ما میتوانیم در هر کجا از برنامه که این تابع فراخوانی شد اکشن هایمان را به نمایش در بیاوریم. به این کار سکشن بندی میگوند.



فایل های خارجی css و js هرکدام در قسمت مخصوصی از کد دریافت میشنود.در طراحی یک صفحه وب فایل css در بالا صفحه (<head>) و فایل های js در انتهای دستورات دریافت میشوند.

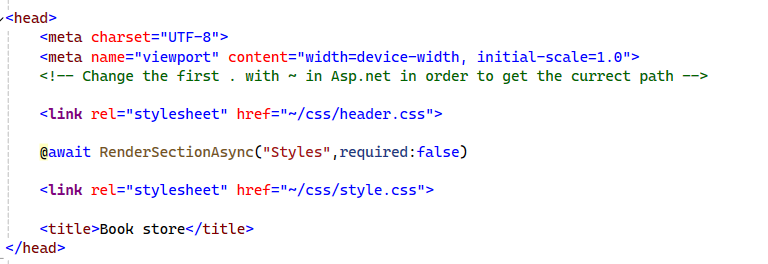
تابع دیگری به نام RenderSectionAsync وجود دارد که به سکشن بندی کمک میکند. محتویات section با استفاده از یک helper به نام **RenderSection**  در Layout قرار می گیرد. این helper دو پارامتر می گیرد. پارامتر اول نام section هست که برای صدا زدن در صفحات پیرو ( view هایی که از این Layout استفاده می کنند) استفاده می شود.

پارامتر دوم یک مقدار true/false هست که می گوید آیا این section باید در تمامی صفحات پیرو استفاده شود یا خیر. مقدار پیش فرض برای این پارامتر true هست و درصورتی که مقداری برای این پارامتر مشخص نکنید true منظور می شود. اگر این پارامتر را برابر با مقدار true قرار دهید در صورتی که در صفحات پیرو این section را صدا نزنید با خطا مواجه می شوید.

@await RenderSectionAsync("Styles",required:false)

* میخواهیم اسلایدر و پروداکت در صفحه اول دیده شوند. برای سئو بهتر و سایتی حرفه ای تر، هر فایل خارجی ای باید در صفحه مخصوص خودش لینک شود.

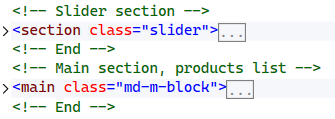
برای مثال: ما در پوشه css خود چند فایل با نام style و slider و product داشتیم.فایل style در قالب توسط تگ link دریافت میشود. اما slider و product را در HomeController توسط مفهوم سکشن دریافت میکنیم. و با استفاده از RenderSectionAsync در هر کجای قالب نام سکشن مورد نظر را صدا میزنیم.

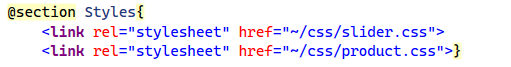


آدرس دهی فایل های css به صورت زیر است:

<**link** rel="stylesheet" href="~/css/style.css">

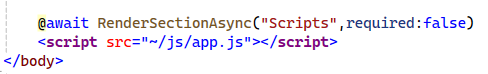
* به view اکشن متد index در HomeController می رویم و دستورات مربوط به اسلایدر و پروداکت را از \_MainLayout به این صفحه انتقال میدهیم.



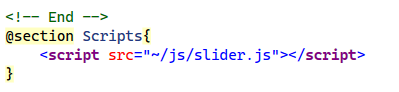
* در همین صفحه یک سکشن با نام Styles میسازیم و در آن فایل های css اسلایدر و پروداکت را لینک میکنیم. نام این سکشن باید با نامی که در RenderSectionAsync ایجاد کردیم یکی باشد.

فایل های جاوا اسکریپت مربوط به پروداکت و اسلایدر نیز باید در \_MainLayout توسط RenderSectionAsync استفاده شوند و در ویو index سکشن بندی شوند.

* به \_MainLayout رفته و در انتهای دستورات فایل جاوا اسکریپت را فراخوانی کنید:



سپس سکشن مربوطه را در view اکشن متد index در HomeController در انتهای دستورات اضافه کنید.

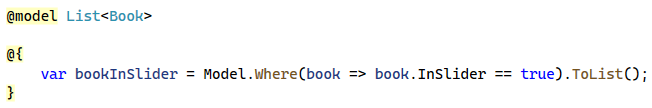


💡) به فایل style رفته و آدرس دهی را از . به .. تغییر دهید. در فایل های css نمیتوان از علامت ~ استفاده کرد.

در HomeController ما لیستی از کتاب ها را دریافت کردیم تا آنها را در view اکشن متد index نمایش دهیم. اطلاعات دوباره باید توسط دستور @model در view دریافت شود.

برای ارسال اطلاعات به index مراحل زیر را طی میکنیم:

* به فایل index بروید.



در مرحله اول لیستی از جنس کتاب دریافت می شود. سپس با کد C# چک میکنیم که آیا در لیست کتابی هست که خصوصیت InSlaider آن فعال باشد؟

InSlaider در مدل به عنوان یک خصوصیت از جنس bool تعریف شده است. دستور بالا تمام کتاب هایی با خاصیت InSlaider را در یک لیست میریزد.

**ساخت صفحه Create :**

* به BookController بروید و برای متد Create یک view بسازید با نام خودش.

از فایل های کمکی فایل Create.html را باز کنید و تگ main آن را در view خود کپی کنید.سپس در ابتدا فایل view باید مدلمان را دریافت کنیم.

@model Book

ما برای اضافه کردن به پایگاه داده نیاز به لیست اطلاعاتی از کتاب ها نیاز نداریم و فقط میخواهیم یک کتاب را دریافت کنیم. پس از Book استفاده میکنیم.

💡) در صورت لزوم اگر عکس یا فایلی لینک شده آدرس آن را از . به ~ تغیر دهید.

میخواهیم در این صفحه فرمی نمایش دهیم و از کاربر مقادیری مانند نام کتاب ، نام ناشر و .. را دریافت کنیم.

<**form** method="post" **asp-controller**="Book" **asp-action**="Create" >

💡) تگ های input تگ هایی هستند برای دریافت ورودی از کاربر.

برای اینکه تگ input بداند که مقدار دریافتی متعلق به کدام prop است از دستور زیر استفاده میشود:

<label **asp-for**="title">title\*</label>

<**input** **asp-for**="Title" id="title" **type**="text" placeholder="Exp: stars in the sky">

<**span** **asp-validation-for**="Title"></**span**>

به همین ترتیب می توان مابقی prop ها را مقداردهی کرد .

Placeholder مقدار پیش فرضی است که در صورت خلی بودن TextBox به ما مقدار داده شده را نمایش میدهد.

**تگ هلپر های asp**

تگ هلپرها (Tag Helpers) در [ASP.NET](https://asp.net/) Core ابزاری هستند که به توسعه‌دهندگان اجازه می‌دهند تا HTML را به صورت داینامیک و با استفاده از سینتکس C# تولید کنند.

این ابزارها به شما این امکان را می‌دهند که کدهای HTML خود را به صورت خواناتر و قابل نگهداری‌تر بنویسید. تگ هلپرها معمولاً در فایل‌های Razorبا پسوند .cshtml استفاده می‌شوند

ویژگی‌های کلیدی تگ هلپرها:

1. سینتکس ساده: تگ هلپرها از سینتکس HTML استفاده می‌کنند و به همین دلیل یادگیری و استفاده از آن‌ها آسان است
2. قابلیت استفاده مجدد: می‌توانید تگ هلپرهای سفارشی خود را ایجاد کنید و در پروژه‌های مختلف از آن‌ها استفاده کنید
3. پشتیبانی از الگوهای MVC: تگ هلپرها به خوبی با الگوهای MVC ادغام می‌شوند و می‌توانند به راحتی داده‌های مدل را نمایش دهند

انواع تگ هلپرها:

1. تگ هلپرهای استاندارد:

<input> برای ایجاد فیلدهای ورودی مختلف مانند متن، چک باکس، دکمه رادیویی و غیره.

<form> برای ایجاد فرم‌ها.

<a> برای ایجاد لینک‌ها.

2. تگ هلپرهای خاص:

<label> برای ایجاد برچسب‌های مرتبط با فیلدهای ورودی.

<select> برای ایجاد منوهای کشویی.

در زیر چند مثال از تگ هلپرها آورده شده است:

استفاده از تگ هلپر <input>

<form **asp-action** ="Create" method="post">

<label **asp-for** ="Name"></label>

<input **asp-for** ="Name" />

<span **asp-validation-for** ="Name"></span>

<input type="submit" value="Submit" />

</form>

استفاده از تگ هلپر <select>

<select asp-for="SelectedOption" asp-items="Model.Options">

<option value="">Select an option</option>

</select>

ایجاد تگ هلپر سفارشی

شما می‌توانید تگ هلپرهای سفارشی نیز بسازید. برای این کار باید یک کلاس جدید ایجاد کنید که از TagHelper ارث‌بری کند و متد Process را پیاده‌سازی کنید.

[HtmlTargetElement("my-tag-helper")]

public class MyTagHelper : TagHelper

{

public string Text { get; set; }

public override void Process(TagHelperContext context, TagHelperOutput output)

{

output.TagName = "div"; // تغییر نام تگ

output.Content.SetContent(Text); // تنظیم محتوا

}

}

در ASP.NET Core، این ویژگی‌ها (attributes) به شما کمک می‌کنند تا به سادگی و با استفاده از تگ هلپرها، فرم‌ها و عناصر HTML را به داده‌های مدل و اکشن‌های کنترلر متصل کنید. بیایید هر یک از آن‌ها را توضیح دهیم:

**تگ هلپر asp-action**

توضیح: این ویژگی مشخص می‌کند که فرم یا لینک به کدام اکشن (عملکرد) در کنترلر متصل شود.

استفاده: معمولاً در تگ <form> یا <a> استفاده می‌شود.

مثال:

<form **asp-action** ="Create" method="post">

<!-- فیلدهای فرم -->

</form>ش

در این مثال، وقتی کاربر فرم را ارسال کند، درخواست به اکشن Create در کنترلر مربوطه ارسال می‌شود.

**asp-controllerتگ هلپر**

توضیح: این ویژگی مشخص می‌کند که کدام کنترلر باید اکشن مربوطه را فراخوانی کند.

استفاده: معمولاً در تگ <form> یا <a> استفاده می‌شود.

مثال:

<form asp-controller="Home" **asp-action** ="Create" method="post">

<!-- فیلدهای فرم -->

</form>

در این مثال، فرم به اکشن Create در کنترلر Home ارسال می‌شود.

**asp-forتگ هلپر**

توضیح: این ویژگی برای ارتباط دادن یک عنصر HTML مانند <input>, <label>, و غیره با یک خاصیت خاص از مدل استفاده می‌شود. این کار باعث می‌شود که فیلدها به صورت خودکار با داده‌های مدل پر شوند و همچنین اعتبارسنجی‌ها را نیز مدیریت کنند.

استفاده: معمولاً در تگ‌های ورودی و برچسب‌ها استفاده می‌شود.

مثال:

<label **asp-for** ="Name"></label>

<input **asp-for** ="Name" />

در اینجا، برچسب و فیلد ورودی به خاصیت Name از مدل متصل شده‌اند. هر تغییری در مدل به‌طور خودکار در این فیلدها منعکس می‌شود.

**asp-validation-forتگ هلپر**

توضیح: این ویژگی برای نمایش پیام‌های اعتبارسنجی مرتبط با یک خاصیت خاص از مدل استفاده می‌شود. اگر اعتبارسنجی در سمت سرور یا کلاینت شکست بخورد، پیام خطا به‌طور خودکار در اینجا نمایش داده می‌شود.

استفاده: معمولاً در تگ‌های <span> یا <div> استفاده می‌شود.

مثال:

<span **asp-validation-for** ="Name"></span>

در این مثال، اگر اعتبارسنجی برای خاصیت Name شکست بخورد، پیام خطا در این تگ نمایش داده می‌شود.

**تمرین:**

ساخت Todo App با داشتن ویژگی نمایش Todo ها بر اساس اولویت، به عنوان مثال اون وظیفه ای(Todo) که اولویت بالاتری داره اول نمایش داده بشه.

ویژگی های تمرین:

● ساخت مدل Todo

● داشتن اولویت های High, Medium ,Low

● وصل شدن به پایگاه داده

● انجام عملیات  CRUD ایجاد، خواندن، ادیت، حذف

● فیلتر کردن بر اساس اولویت Todo ها

○ توجه: نمایش لیست todo ها در صفحه index تودوها بر اساس تاریخ ساخته شدن، یعنی تودویی که جدیدتر ساخته میشه باید بالای جدول باشه.

نمایش لیست todo ها در صفحه اول بر اساس اولویت داده شده.

● نمایش Todos در صفحه اول (در  Home/index)

● شرطی رندر کردن نمایش نوشته "کامل شده" در صورت کامل کردن اون Todo زدن تیک کامل شدن

● نمایش پیغام "با موفقیت ثبت/حذف/ادیت شد"

○ توجه:

• طراحی و استفاده از ایکون ها اختیاری است اما حتما باید برای اینکه مشخص بشه کدام todo کامل شده یا نه از شرطی رندر کردن استفاده بشه و پیام کامل شده یا کامل نشده رو نمایش دهد.

* در مرحله اول برای پروژه یک مدل با مشخصات زیر می سازیم:

[Key]

public int Id { get; set; }

[Required(ErrorMessage = "Please fill the required filed!")]

[MaxLength(15)]

public string Title { get; set; }

public int TaskPriority { get; set; }

public DateTime CreatedAt { get; set; } = DateTime.Now;

public bool IsCompleted { get; set; }

[Required(ErrorMessage = "Please fill the required field!")] یک ویژگی (attribute) در ASP.NET Core است که برای اعتبارسنجی مدل‌ها استفاده می‌شود.

این ویژگی به شما این امکان را می‌دهد که مشخص کنید یک فیلد خاص در مدل باید پر شود و در صورت عدم پر شدن، پیام خطای مشخصی نمایش داده شود.

[Required] این ویژگی نشان می‌دهد که فیلد مرتبط با آن الزامی است و باید حتماً مقداری داشته باشد. اگر کاربر این فیلد را خالی بگذارد، اعتبارسنجی شکست می‌خورد.

ErrorMessage این پارامتر به شما این امکان را می‌دهد که پیام خطای سفارشی برای این فیلد تعیین کنید. اگر اعتبارسنجی شکست بخورد، این پیام به کاربر نمایش داده می‌شود.

▎مثال

فرض کنید یک مدل به نام User دارید که شامل یک خاصیت Name است. می‌توانید از [Required] به شکل زیر استفاده کنید:

public class User

{

[Required(ErrorMessage = "Please fill the required field!")]

public string Name { get; set; }

// سایر خاصیت‌ها

}

▎نحوه استفاده در فرم

در فرم HTML ، اگر کاربر سعی کند فرم را بدون پر کردن فیلد Name ارسال کند، پیام خطای مشخص شده نمایش داده می‌شود:

<form asp-action="Create" method="post">

<label asp-for="Name"></label>

<input asp-for="Name" />

<span asp-validation-for="Name"></span>

<button type="submit">Submit</button>

</form>

[MaxLength(15)] یک ویژگی در [ASP.NET](https://asp.net/) Core است که برای محدود کردن طول یک فیلد متنی (مثل رشته) به حداکثر ۱۵ کاراکتر استفاده می‌شود. اگر کاربر سعی کند بیشتر از ۱۵ کاراکتر وارد کند، اعتبارسنجی شکست می‌خورد و پیام خطا نمایش داده می‌شود.

* سپس کنترلری با نام todo ساخته و برای نمایش کار ها بر اساس زمان از دستورات زیر استفاده می کنیم.

public IActionResult Index()

{

var Todo = \_context.Todos.OrderByDescending(x => x.CreatedAt).ToList();

return View(Todo);

}

تحلیل کد:

نوع بازگشتی:

IActionResult این نوع نشان‌دهنده نتیجه‌ای است که می‌تواند به مرورگر ارسال شود. این نتیجه می‌تواند شامل نمای (View)، ریدایرکت، یا دیگر انواع پاسخ‌ها باشد.

دریافت داده‌ها:

\_context.Todosاین بخش به دیتابیس دسترسی دارد و مجموعه‌ای از اشیاء Todo را از آن دریافت می‌کند. \_context معمولاً یک شیء از نوع DbContext است که برای تعامل با پایگاه داده استفاده می‌شود.

مرتب‌سازی:

OrderByDescending(x => x.CreatedAt) این قسمت لیست Todo ها را بر اساس تاریخ ایجاد (CreatedAt) به ترتیب نزولی مرتب می‌کند. به عبارت دیگر، جدیدترین موارد در بالای لیست قرار می‌گیرند.

تبدیل به لیست:

ToList()این متد نتایج مرتب‌شده را به یک لیست تبدیل می‌کند. این کار باعث می‌شود که داده‌ها به صورت یک مجموعه قابل استفاده در کد بعدی درآیند.

بازگشت به نمای:

return View(Todo) در نهایت، داده‌های Todo به نمای مربوطه ارسال می‌شوند تا در صفحه وب نمایش داده شوند.

* برای این کنترلر یک view میسازیم و مدل را دریافت می کنیم:

@model List<Todo>

@{

var showPriority = "";

}

متغیری می سازیم که درون آن رشته خالی ای قرار دارد و درون آن رشته وضعیت کارها قرار می گیرد.

<body>

<a href="@Url.Action("Create", "Todo")" class="btns">Add New </a>1

<div >

<table>

<thead>

<tr>

<th>Title</th>

<th>Priority</th>

<th>Is it completed?</th>

<th>Created at</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var task in Model)

{

<tr>

<td>@task.Title</td>

<td>

<h2>

@switch (task.TaskPriority)

{

case 1 :

showPriority = "high";

break;

case 2 :

showPriority = "medium";

break;

case 3 :

showPriority = "low";

break;

}

@showPriority;2

</h2>

</td>

<td>@(task.IsCompleted ? "Completed" : "Not finished")</td>3

<td>@task.CreatedAt.ToShortDateString()</td>

<td>

<div>

<**a** **asp-controller**="Dashboard" **asp-action**="Edit" **asp-route-id**="@task.Id">

<button style="margin: 10px;" class="btns"> Edit </button>

</**a**>

<**form** method="post" **asp-controller**="Dashboard" **asp-action**="Delete" **asp-route-id**="@task.Id" class="flex">4

<button type="submit" class="btns">Delete</button>

</**form**>

</div>

</td>

</tr>

}

</tbody>

</table>

</div>

</body>

1. href="@Url.Action("Create", "Todo")" توسط یک لینک و عبارت ذکر شده پیوندی به متد create ایجاد می کند.
2. کد شما از دستور switch برای تعیین اولویت یک تسک استفاده می‌کند. بر اساس مقدار task.TaskPriority، متغیر showPriority به یکی از سه مقدار ("high"، "medium"، "low") تنظیم می‌شود. اگر مقدار مشخصی نباشد، هیچ عملی انجام نمی‌شود.
3. این عبارت یک شرط را بررسی می‌کند: اگر تسک کامل شده باشد، "Completed" نمایش داده می‌شود؛ در غیر این صورت، "Not finished" نمایش داده می‌شود.
4. در این قطعه کد، asp-route-id یک ویژگی است که به Razor در ASP.NET Core می‌گوید که مقدار @task.Id را به عنوان پارامتر id به آدرس URL درخواست POST اضافه کند. این به سرور کمک می‌کند تا شناسایی کند کدام تسک باید حذف شود.

جهت نمایش لیست todo ها در صفحه اول بر اساس اولویت داده شده به اکشن index در کنترلر Todo بروید:

public IActionResult Index()

{

var Todo = \_context.Todos.OrderByDescending(x => x. TaskPriority).ToList();

return View(Todo);

}

**بخش چهارم (Relationship)**

**انواع رابطه در .net**

رابطه‌های بین جداول در پایگاه داده‌ها به سه نوع اصلی تقسیم می‌شوند: یک به یک، یک به چند و چند به چند. در ادامه هر کدام را به همراه مثال توضیح می‌دهم**:**

**رابطه یک به یک (1:1)**

در این نوع رابطه، هر رکورد در جدول اول با حداکثر یک رکورد در جدول دوم مرتبط است و بالعکس.

مثال: جدول کاربران و جدول پروفایل‌ها: هر کاربر فقط یک پروفایل دارد و هر پروفایل فقط متعلق به یک کاربر است.

**رابطه یک به چند (1:N)**

در این نوع رابطه، هر رکورد در جدول اول می‌تواند با چند رکورد در جدول دوم مرتبط باشد، اما هر رکورد در جدول دوم تنها می‌تواند به یک رکورد در جدول اول مرتبط باشد.

مثال: جدول نویسندگان و جدول کتاب‌ها: هر نویسنده می‌تواند چند کتاب نوشته باشد، اما هر کتاب تنها متعلق به یک نویسنده است.

**رابطه چند به چند (N:M)**

در این نوع رابطه، هر رکورد در جدول اول می‌تواند با چند رکورد در جدول دوم مرتبط باشد و بالعکس.

مثال: جدول دانش‌آموزان و جدول کلاس‌ها: هر دانش‌آموز می‌تواند در چند کلاس ثبت‌نام کند و هر کلاس می‌تواند شامل چند دانش‌آموز باشد.

برای پیاده‌سازی رابطه‌های چند به چند معمولاً از یک جدول واسط استفاده می‌شود که هر دو طرف را به هم متصل می‌کند.

تمرین) میخواهیم ارتباطی یک به چند بین product و category ایجاد کنیم:

هر دسته بندی میتواند شامل چندین محصول شود اما هر محصول فقط میتواند یک دسته بندی داشته باشد . بین محصول و دسته بندی یک رابطه یک به چند وجود دارد.

* در مرحله اول برای محصول یک مدل با مشخصات زیر میسازیم:

public int Id { get; set; }

public string Title { get; set; }

public int CategoryId { get; set; }

//navigate to category

public Category? Category { get; set; }1

* سپس یک مدل هم برای دسته بندی ایجاد میکنیم:

public int Id { get; set; }

public string Name { get; set; }

//navigate to product

public List<Product>? Products { get; set; }2

1. هر محصول می تواند یک دسته بندی داشته باشد پس در مدل product ما یک ویژگی با نام CategoryId و یک ویژگی از جنس مدل category ایجاد می کنیم. از این ویژگی برای ایجاد ارتباط بین جداول استفاده میشود.
2. از آنجایی که در سمت category چندین محصول را میتوانیم داشته باشیم پس میتوان گفت کتگوری لیستی از محصولات را باید دریافت کند.

💡) علامت سوال موجب میشود تا در صورت خالی بودن مقدار متغیر، برنامه دچار خطا نشود.

* سپس در کلاس ApplicationDbContext دو جدول میسازیم:

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

* سپس توسط add-migration جدول را ایجاد میکنیم:

Add-migration proudTb

Update database

💡) برای ایجاد همه جدول ها فقط یک add-migration نیاز است.

* برای کتگوری یک کنترلر میسازیم. میخواهیم لیستی از دسته بندی ها را به نمایش بگذاریم. برای انجام این کار در متد index کنترلر category لیستی از دسته بندی ها را به view پاس میدهیم:

public IActionResult Index()

{

var categories = \_context.Categories.ToList();

return View(categories);

{

* برای index یک view میسازیم و لیست را در ویو دریافت میکنیم و همچنین پیوندی به صفحه create ایجاد می کنیم:

@model List<Category>

<h1>Category index</h1>

<**a** **asp-controller**="Category" **asp-action**="Create" >Create</**a**>

@foreach(var cat in Model)

{

<h2>@cat.Name</h2>

{

* سپس در کنترلر متد create را ایجاد کرده برای آن یک view بسازید.

@model Category

<h1>Create</h1>

<**form** method="post" **asp-action**="Create">

<label>Category Items:</label>

<**input** **asp-for**="Name" />

<button type="submit">add category</button>

* مدل product هم به یک کنترلر هم نام خود نیاز دارد. کنترلر را ساخته و جدول را دریافت می کنیم. سپس میخواهیم در صفحه اصلی به ما لیستی از محصولات و دسته بندی آنها نمایش داده شود.

public IActionResult Index()

{

var products = \_context.Products.Include(q => q.Category).ToList();

return View(products);

}

به طور خلاصه، این کد تمام محصولات را از پایگاه داده به همراه اطلاعات مربوط به دسته‌بندی آن‌ها بارگذاری می‌کند و نتیجه را به صورت یک لیست برمی‌گرداند.

توضیحات:

\_context.Products به مجموعه محصولات در پایگاه داده اشاره دارد.

Include(q => q.Category) مشخص می‌کند که اطلاعات دسته‌بندی مربوط به هر محصول نیز بارگذاری شود.

ToList() نتایج را به یک لیست تبدیل می‌کند.

به این ترتیب، با یک بار فراخوانی، هم محصولات و هم دسته‌بندی‌های آن‌ها را دریافت می‌کنید.

💡) از آنجایی که ما ویژگی ای با نام category در product داریم پس اطلاعات دسته بندی در محصولات هست.

* برای نمایش محصولات به view اکشن index می رویم و لیستی از محصولات را دریافت میکنیم:

@model List<Product>

<h1>Product index</h1>

<**a** **asp-controller**="Product" **asp-action**="create">Create</**a**>

@foreach (var prod in Model)

{

<h2>@prod.Title</h2>

<h4>@prod.Category?.Name</h4>

}

این دستور لیستی از محصولات و دسته بندی آنها نمایش میدهد. @prod.Category?.Name این دستور به ویژگی های کتگوری دسترسی پیدا میکند و به خاطر وجود ؟ در صورت خالی بودن دچار خطایی نمیشود.

* برای اضافه کردن به محصولات ما نیاز به اکشن create و صفحه ای برای ایجاد محصولات داریم. میخواهیم در زمان ایجاد محصول از لیست کتگوری نوع محصول را انتخاب کنیم:

به کنترلر product می رویم و دستورات زیر را اضافه می کنیم:

public IActionResult Create()

{

ViewBag.category = new SelectList(\_context.Categories.ToList(), "ID", "Name");

return View();

}

[HttpPost]

public IActionResult Create(Product prod)

{

if (ModelState.IsValid)

{

\_context.Products.Add(prod);

\_context.SaveChanges();

return RedirectToAction("Index");

}

return View();

}

این کد یک اکشن در کنترلر [ASP.NET](https://asp.net/) Core است که برای نمایش فرم ایجاد یک محصول استفاده می‌شود. در این اکشن ViewBag.category یک لیست انتخابی از دسته‌بندی‌ها را ایجاد می‌کند SelectList برای پر کردن یک Dropdown در ویو استفاده می‌شود، با استفاده از شناسه و نام دسته‌بندی‌ها و در نهایت، ویو مربوطه بازگشت داده می‌شود.

ViewBag یک ویژگی دینامیک است که برای ارسال داده‌ها از کنترلر به ویو استفاده می‌شود. در اینجا، ما از آن برای ارسال لیست دسته‌بندی‌ها به ویو استفاده می‌کنیم.

ViewBag.category به عنوان یک لیست انتخابی (SelectList) تنظیم می‌شود

SelectList یک کلاس در [ASP.NET](https://asp.net/) است که برای ایجاد لیست‌های انتخابی (Dropdown) استفاده می‌شود.

context.Categories به مجموعه‌ای از دسته‌بندی‌ها اشاره دارد که از پایگاه داده بازیابی شده‌اند "Id" و "Name" به ترتیب مشخص‌کننده‌ی فیلدهای شناسه و نام دسته‌بندی‌ها هستند. این بدان معناست که در Dropdown، مقدار هر گزینه برابر با Id و متن نمایش داده شده برابر با Name خواهد بود.

به طور خلاصه این کد یک لیست انتخاب از category را ایجاد میکند و آن را در ViewBag ذخیره می کند تا بتوانیم در view از آن استفاده کنیم.

* برای create یک view می سازیم.

@model Product

<h1>Create product</h1>

<**form** method="post" **asp-controller**="product" **asp-action**="Create">

<label>Product name:</label>

<**input** **asp-for**="Title"/>

<**select** **asp-for**="CategoryId">

@foreach (var category in ViewBag.category)

{

<**option** **value**="@category.Id">@category.Name</**option**>

}

</**select**>

<button type="submit">add</button>

</**form>**

رابطه یک به چند بین محصول و دسته‌بندی به این معناست که هر محصول می‌تواند به یک دسته‌بندی خاص تعلق داشته باشد، اما هر دسته‌بندی می‌تواند شامل چندین محصول باشد. در این رابطه، CategoryId در مدل محصول به عنوان کلید خارجی برای اشاره به شناسه دسته‌بندی استفاده می‌شود. این ساختار به ما اجازه می‌دهد تا محصولات را بر اساس دسته‌بندی‌های مختلف سازماندهی و مدیریت کنیم.

از تگ select در اینجا برای ایجاد یک لیست استفاده شده و درون آن توسط یک حلقه به ازای هر category در ViewBag.category یک option بر اساس id مدل کتگوری نام آن را نمایش میدهد.

تمرین)

**بخش پنجم (area)**

با ایجاد یک Area می توان کنترلر ها و اکشن متد هارا وابسته به کارکردشان جدا کرد. برای مثال در سایت ممکن است پنلی جدا برای ادمین داشته باشیم.

یک area برای جدا سازی بخش user و admin داریم. برای ساخت Area مسیر زیر را بروید:

Add> new scaffold > new area > name

سپس اسمی برای area انتخاب کنید. پس از ساخته شدن یک note برای شما ارسال میشودکه باید آن را در program.cs وارد کنید.

💡) همه پکیج ها را به اخرین نسخه آپدیت کنید تا به خطا بر نخورید.

سپس یک map controller routeدر program.cs ایجاد کنید:

app.MapControllerRoute(name: "area",

pattern: "{area= Area name}/{controller=controller name}/{action=Index}");

سپس وارد کنترلر مورد نظر شوید و دستور زیر را در ابتدای کنترلر بنویسید:

[Area(“Area name”)]

حالا کافی است در مرورگر آدرس زیر را وارد کنید:

Area name / Controller name / action method

💡) برای اینکه layout اصلی بر روی view های area اعمال شود توی فایل \_ViewStart.cshtml که داخل Area ساختیم (مثلاً /Areas/Admin/Views/) دستور زیر را قرار میدهیم:

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

حالا هر ویویی که تو Area هست، به صورت اتوماتیک از Layout اصلی پروژه پیروی می‌کند.

**بخش ششم (عکس ها در .net)**

می خواهیم لیستی از دانشجو ها داشته باشیم و هر دانشجو عکس خودش را داشته باشد. مدل دانشجو باید شامل ویژگی های زیر باشد:

public int Id { get; set; }

public string Name { get; set; }

public string picName { get; set; }

سپس به کنترلر دانشجو رفته و دستورات زیر را وارد میکنیم:

private readonly ApplicationDbContext \_context;

public StudentController(ApplicationDbContext context)

{

\_context = context;

}

private readonly IWebHostEnvironment \_webHostEnvironment;

public StudentController(IWebHostEnvironment webHostEnvironment)

{

\_webHostEnvironment = webHostEnvironment;

}

دستور اول برای ساخت جدول و نمونه قاب دسترسی ای از آن و دستور دوم نیز به ما امکان دسترسی به مسیرهارا می دهد. برای مثال برای دسترسی به پوشه روت میتوان از این متغیر استفاده کرد.

پوشه WWWroot پوشه ای است که در کل برنامه قابل دسترسی است و فایل های خارجی مثل عکس درون آن قرار می گیرند. به این پوشه رفته و فایلی به نام pic بسازید.

در مرحله بعد کنترلری هم نام مدل ساخته و در آن اکشن متدی با نام Addstu می سازیم که کار این تابع اضافه کردن دانشجو و عکس آن به پایگاه داده است و ورودی آن یک دانشجو و یک فایل (عکس) است.

public IActionResult AddStu()

{

return View();

}

[HttpPost]

public IActionResult AddStu(Student st , IFormFile file)

{

string rootpath = \_webHostEnvironment.WebRootPath;1

if (file != null)

{

string FileName = Guid.NewGuid().ToString() +Path.GetExtension(file.FileName);2

string StuImagePath = Path.Combine(rootpath, @"pic\stu");3

using (var FileStream = new FileStream(Path.Combine(StuImagePath, FileName), FileMode.Create)) ;4

st.picName = @"pic\stu\" + FileName;5

}

\_context.Students.Add(st);

\_context.SaveChanges();

return RedirectToAction("Index");

}

1. string rootpath = \_webHostEnvironment.WebRootPath;

مسیر ریشه وب‌سایت را به متغیر rootpath نسبت می‌دهد.

2. string FileName = Guid.NewGuid().ToString() + Path.GetExtension(file.FileName);

یک نام فایل منحصر به فرد ایجاد می‌کند که شامل یک Guidو پسوند فایل است. در قسمت اول Guid یک فایل ایجاد کرده و توسط قسمت دوم پسوند فایل آپلود شده را دریافت کرده و به قسمت اول می چسبانیم.

3. string StuImagePath = Path.Combine(rootpath, @"pic\stu");

مسیر کامل پوشه‌ای که تصاویر دانش‌آموزان در آن ذخیره می‌شود را ایجاد می‌کند. Rootpath مسیرwwwroot است و @"pic\stu" آدرس پوشه عکس ها که توسط combine این دو ترکیب شده در StuImagePath ریخته می شود.

4. using (var FileStream = new FileStream(Path.Combine(StuImagePath, FileName), FileMode.Create)) ;

یک جریان فایل برای ذخیره تصویر در مسیر مشخص شده ایجاد می‌کند . توسط دستور using فایلی با نام و پسوند گفته شده با مود create ساخته میشود و فایل آپلود شده در آن کپی می شود.

5. st.picName = @"pic\stu\" + FileName;

نام فایل تصویر را به ویژگی picName شیء Student نسبت می‌دهد تا در پایگاه داده ذخیره شود. حالا نام فایل و آدرس آن را به picName که متغیری از جنس رشته بود میدهیم.

و در مرحله آخر نمونه ای که از دانشجو ساختیم با نام st که شامل اسم و نام عکس میشود را به پایگاه داده با add اضافه میکنیم.

سپس باید یک view برای دریافت اطلاعات کاربر از جمله عکس بسازیم. پس برای اکشن AddStu یک ویو میسازیم:

@model Student

<h1>Add Student</h1>

<**form** method="post" **asp-action**="AddStu" enctype="multipart/form-data">

<div>

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name"/>

</div>

<div>

<label>PicName</label>

<**input** **type**="file" **asp-for**="picName" />

</div>

</**form**>

**بخش هفتم (view model)**

یک الگو طراحی است برای جدا کردن داده ها . برای مثال میتوانید از میان خصوصیات یک کلاس فقط یک دسته از آن ها را لیست کنید تا در view نمایش دهید.

برای انجام این کار میتوانید مدلی بدون جدول بسازید که وابسته به مدل اصلی است. به این مدل به اصطلاح view model می گویند.

تمرین) یک جدول از مشخصات دانشجو ها بسازید و در یک کمبو باکس اسامی را به نمایش در آورید.

در این مسئله دانشجو می تواند خصوصیاتی مثل ورودی ، دروس ، شهریه و غیره را داشته باشد. ولی ما فقط به دنبال لیستی از اسامی هستیم و نیاز به مابقی خصوصیات نداریم. برای کوچک تر کردن مسئله مدلی می سازیم که دارای یک خصوصت با نام عضو است که اسامس در ان قرار می گیرند و خصوصیت دیگر یک لیست از SelectListItem است که لیستی از آیتم های کمبو باکس را نگه داری میکند.

* در مرحله اول مدل اصلی را می سازیم:

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

}

* در مرحله دوم ویو مدلی با جنس لیست آیتم میسازیم تا در ادامه در کنترلر دسته بندی آن را مشخص کنیم:

public class ViewModel1

{

public string member { get; set; }

public List<SelectListItem> members { get; set; }

}

یک خصوصیت از جنس رشته و یک خصوصیت دیگر از جنس لیست که اعضای آن SelectListItem است.

این دو ویژگی در اصل یکی هستند و در کنار هم می آیند.

* سپس در مرحله بعد برای دریافت لیست اسامی دانشجویان به کنترلر دانشجو میرویم:

public IActionResult Index()

{

ViewModel1 model1 = new ViewModel1();1

var item = \_Context.Students.Select(x=> new SelectListItem {

Value = x.Id.ToString(), Text = x.Name, } ).ToList();2

model1.members = item;3

return View(model1);

}

1. از ViewModel1 یک نمونه می سازیم .
2. توسط تابع Select میتوانیم بخشی از جدول را لیست کنیم. سپس با SelectListItem میتوانیم مشخصه ای را که در مدل اصلی است در کمبو باکس به نمایش درآورید.

SelectListItem شامل دو بخش Value و Text می شود. Value مقدار کلید است و Text مقدار وابسته به کلید است.

1. سپس در نمونه ای که ساختیم به سراغ خصوصیت members میرویم و نتایج select را در آن میریزیم.

* برای نمایش لیست یک ویو برای متد می سازیم:

@model ViewModel1

<**select** **asp-for**="member" **asp-items**="Model.members"></**select**>

**بخش هشتم (ارسال اطلاعات از view به کنترلر)**

* ارسال داده توسط asp-route

asp-route یک ویژگی (attribute) در ASP.NET Core MVC و Razor Pages است که به شما این امکان را می‌دهد تا پارامترهای مسیر (route parameters) را به لینک‌ها و فرم‌ها اضافه کنید. این ویژگی به شما کمک می‌کند تا URLهای دینامیک بسازید و مقادیر پارامترها را به راحتی در URLها قرار دهید.

استفاده از asp-route  
1. در لینک‌ها  
شما می‌توانید از asp-route در تگ <a> برای ایجاد لینک‌هایی با پارامترهای دینامیک استفاده کنید:

a asp-controller="Home" asp-action="Details" asp-route-id="@item.Id"

در این مثال، asp-route-id مقدار Id را به URL اضافه می‌کند. اگر [item.Id](https://l.ble.ir/?l=https%3A%2F%2Fitem.Id&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) برابر با 5 باشد، URL نهایی چیزی شبیه به /Home/Details/5 خواهد بود.  
2. در فرم‌ها  
شما همچنین می‌توانید از asp-route در تگ <form> برای ارسال پارامترها استفاده کنید:

<form asp-controller="Home" asp-action="Edit" asp-route-id="@item.Id" method="post"> <!-- فیلدهای فرم --> <button type="submit">ویرایش</button></form>  
  
در اینجا، هنگام ارسال فرم، id به URL اضافه خواهد شد.

حالا که نام به کنترلر پاس داده شده در کنترلر آن را دریافت میکنیم:

public IActionResult Index(string Name){…}

* ارسال لیست داده ها به کنترلر توسط asp-all-route-data

asp-all-route-data یک ویژگی (attribute) در [ASP.NET](https://l.ble.ir/?l=https%3A%2F%2FASP.NET&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) Core است که به شما این امکان را می‌دهد تا تمام داده‌های مسیر (route data) را به یک لینک یا فرم اضافه کنید. این ویژگی به ویژه زمانی مفید است که شما بخواهید تمام پارامترهای موجود در مسیر فعلی را در یک لینک جدید یا فرم حفظ کنید.  
1. در لینک‌ها  
شما می‌توانید از asp-all-route-data در تگ <a> برای ایجاد لینک‌هایی استفاده کنید که تمام پارامترهای مسیر فعلی را حفظ کنند:

<a asp-controller="Home" asp-action="Index" asp-all-route-data="@ViewData["RouteData"]">

در این مثال، تمام پارامترهای موجود در RouteData به URL جدید اضافه می‌شوند.

شما همچنین می‌توانید از asp-all-route-data در تگ formبرای ارسال تمامی پارامترها به سرور استفاده کنید:

<form asp-controller="Home" asp-action="Search" asp-all-route-data="@ViewData["RouteData"]" method="post">

<input type="text" name="query" />

<button type="submit">جستجو</button></form>

ویا می توان یک دیکشنری را انتقال داد:

@{ var x = new Dictionary <string,string>{

{“id”,”123’},{“name”,”asal”}}};

<a asp-action=”index” asp-all-route-data=”x”>

سپس دیکشنری ارسال شده را دریافت می کنیم:

public IActionResult Index(Dictionary <string,string x >){…}

**بخش نهم (partial)**

تگ partial در ASP.NET (به ویژه در ASP.NET MVC و ASP.NET Core) برای نمایش قسمت‌های مشترک یا قابل استفاده مجدد از یک ویو (View) استفاده می‌شود. این تگ به شما این امکان را می‌دهد که بخش‌هایی از کد HTML را که ممکن است در چندین ویو تکرار شوند، به صورت جداگانه تعریف کرده و سپس در ویوهای دیگر به کار ببرید.

1. تعریف یک Partial View

ابتدا شما باید یک Partial View بسازید. به عنوان مثال، فرض کنید که یک Partial View به نام \_Header.cshtml دارید:

<header> … </header>

برای ساخت این view به پوشه shared رفته و یک view بسازید و تیک partial را بزنید. نامی که به این view میدهید نام partial شماست.

2. استفاده از Partial View در یک View دیگر

سپس می‌توانید این Partial View را در یک View دیگر با استفاده از تگ partial فراخوانی کنید:

@{ ViewData["Title"] = "صفحه اصلی";}  
<div> @Html.Partial("\_Header") </div>

<partial name=”\_partial1”> … </partial>

نام صفحه ساخته شده در name=”\_partial1” قرار میگیرد.   
3. استفاده از RenderPartial  
شما همچنین می‌توانید از RenderPartial به جای [Html.Partial](https://l.ble.ir/?l=https%3A%2F%2FHtml.Partial&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) استفاده کنید. تفاوت اصلی این است که RenderPartial محتوا را مستقیماً به خروجی می‌نویسد و هیچ مقداری را برنمی‌گرداند:

@{ Html.RenderPartial("\_Header"); }

**بخش دهم (Razor Page)**

Razor Page یک ویژگی در [ASP.NET](https://l.ble.ir/?l=https%3A%2F%2FASP.NET&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) Core است که به توسعه‌دهندگان این امکان را می‌دهد تا صفحات وب را به صورت ساده و کارآمد بسازند. Razor Pages به عنوان یک الگوی برنامه‌نویسی جدید در [ASP.NET](https://l.ble.ir/?l=https%3A%2F%2FASP.NET&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) Core معرفی شده و به طور خاص برای ساخت اپلیکیشن‌های وب با تمرکز بر روی صفحات و تعاملات کاربر طراحی شده است.

**ویژگی‌های اصلی Razor Pages**  
1. ساده‌سازی ساختار پروژه:

Razor Pages به شما اجازه می‌دهد تا منطق و نمای (View) مربوط به هر صفحه را در یک فایل ترکیب کنید، که این امر باعث کاهش پیچیدگی می‌شود.  
2. مدل‌سازی بهتر:

هر Razor Page معمولاً دارای یک فایل .cshtml (که شامل HTML و کد Razor است) و یک فایل [.cshtml.cs](https://l.ble.ir/?l=https%3A%2F%2F.cshtml.cs&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) (که شامل منطق سرور است) می‌باشد. این ساختار به توسعه‌دهندگان کمک می‌کند تا منطق صفحه را به راحتی مدیریت کنند.  
3. مسیر‌یابی آسان:

Razor Pages به طور خودکار مسیرها را بر اساس ساختار دایرکتوری فایل‌ها ایجاد می‌کند. به عنوان مثال، اگر شما فایلی به نام [About.cshtml](https://l.ble.ir/?l=https%3A%2F%2FAbout.cshtml&spec=eyJzcCI6Ijk5MTA0OTE1MyIsImNwIjoiOTkxMDQ5MTUzIn0%3D" \t "_blank) در پوشه Pages داشته باشید، به طور خودکار به URL /About متصل می‌شود.  
4. پشتیبانی از مدل‌ها:

Razor Pages به راحتی می‌تواند با مدل‌ها کار کند، که این امر برای پردازش داده‌ها و ارتباط با پایگاه داده بسیار مفید است.

**نحوه ایجاد یک Razor Page**1. ایجاد پروژه ASP.NET Core: ابتدا یک پروژه ASP.NET Core جدید ایجاد کنید.  
2. ایجاد Razor Page: در پوشه Pages، یک فایل جدید با پسوند .cshtml ایجاد کنید. به عنوان مثال، Contact.cshtml.

@page

@model ContactModel

<h2>تماس با ما</h2>

<form method="post">

<label for="name">نام:</label>

<input type="text" id="name" name="Name" />

<label for="email">ایمیل:</label>

<input type="email" id="email" name="Email" />

<button type="submit">ارسال</button>

</form>

نوشتن منطق در فایل کد پشت:

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.RazorPages;

public class ContactModel : PageModel

{

public void OnGet()

{

// منطق برای GET درخواست

}

public IActionResult OnPost()

{

// منطق برای POST درخواست

// پردازش داده‌ها و ارسال پاسخ

return RedirectToPage("Success");

}}

**Accounting**

فرایند اکانتینگ فرایندی سه مرحله ای است شامل ساخت کاربر و مدیریت آن ، احراز هویت کاربران و مجوز های دسترسی که مایکروسافت برای حل این مسئله سرویسی ارائه داده .

پکیج Microsoft ASP .net core identity entity پکیجی است که به مدیریت کاربران میپردازد و دارای دستورالعمل های آماده است.

1. پس از نصب پکیج ، کلاس ApplicationDbContext را ایجاد می کنیم. این کلاس از DbContext پیروی میکرد. حالا که از پکیج جدید استفاده میکنیم این کلاس باید از عبارت زیر ارث بری کند:

public class ApplicationDbContext : IdentityDbContext<IdentityUser>

{

public ApplicationDbContext(DbContextOptions options) : base(options){}

}

IdentityDbContext : یک کلاس از فریمورک **Entity** Framework Core است که برای مدیریت داده‌های مربوط به هویت و احراز هویتکاربران در برنامه‌های مبتنی بر ASP.NET Core طراحی شده است. این کلاس از DbContext ارث‌بری می‌کند و در پروژه‌هایی که از Identity System استفاده می‌کنند به طور پیش‌فرض برای تعامل با پایگاه داده به کار می‌رود.

IdentityUser جدولی است که خود .net ساخته و متد هایی برای مدیریت کاربران ارائه می دهد.

1. کلاسی به نام user میسازیم که از IdentityUser ارث بری میکند و مدل اصلی است.پکیج را هم اضافه کنید.

Public class User:IdentityUser

{

public int Id { get; set; }

public string Name { get; set; }

}

1. حالا که کلاس user از IdentityUser ارث بری میکند در DbContext به جای استفاده از جدول IdentityUser از user استفاده می کنیم.

Public class ApplicationDbContext : IdentityDbContext<User>

{

public ApplicationDbContext(DbContextOptions options) : base(options)

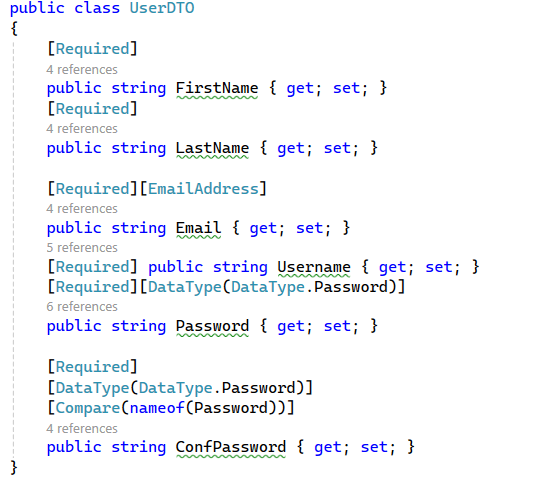
{

}

public DbSet<User> users { get; set; }

}

1. کلاس دیگری می سازیم با نام userDTO که از User ارث بری می کند اما یک جدول نیست.



این‌ها خصوصیات (Attributes) در زبان برنامه‌نویسی سی‌شارپ (C#) هستند که برای مدل‌سازی داده‌ها در ASP.NET استفاده می‌شوند.

[Required]

* مشخص می‌کند که یک فیلد الزامی است و نمی‌تواند خالی باشد

[EmailAddress]

* اعتبارسنجی می‌کند که مقدار واردشده باید یک آدرس ایمیل معتبر باشد

[DataType(DataType.Password)]

* مشخص می‌کند که این فیلد از نوع رمز عبور است. معمولاً برای ارائه تجربه مناسب‌تر در رابط کاربری (مثل مخفی کردن کاراکترها) استفاده می‌شود.

[Compare(nameof(Password))]

* ویژگی [Compare(nameof(Password))] در سی‌شارپ برای اعتبارسنجی استفاده می‌شود و اطمینان می‌دهد که مقدار فیلدی (مثلاً ConfirmPassword) با مقدار فیلد مشخص‌شده (Password) برابر باشد.

این ویژگی‌ها اغلب در مدل‌های MVC برای اعتبارسنجی ورودی‌ها به کار می‌روند.

در اینجا دو کلاس تعریف شده‌اند UserDTO و User هر کدام نقش متفاوتی دارند:

کلاسUserDTO (Data Transfer Object)

این کلاس برای انتقال داده‌ها بین لایه‌های مختلف برنامه استفاده می‌شود، مخصوصاً برای گرفتن داده‌ها از کاربر در فرآیندهایی مانند ثبت‌نام یا ورود به سیستم.

از اعتبارسنجی‌های مختلف مثل [Required] و [EmailAddress] استفاده می‌کند تا مطمئن شود داده‌های ورودی معتبر هستند و شامل اطلاعاتی مثل FirstName, LastName, Email, Username, و Password است.

هدف استفاده از آن کنترل و اعتبارسنجی داده‌های کاربر قبل از ذخیره در پایگاه داده و بهبود امنیت با جداسازی مدل انتقال داده از مدل پایگاه داده.

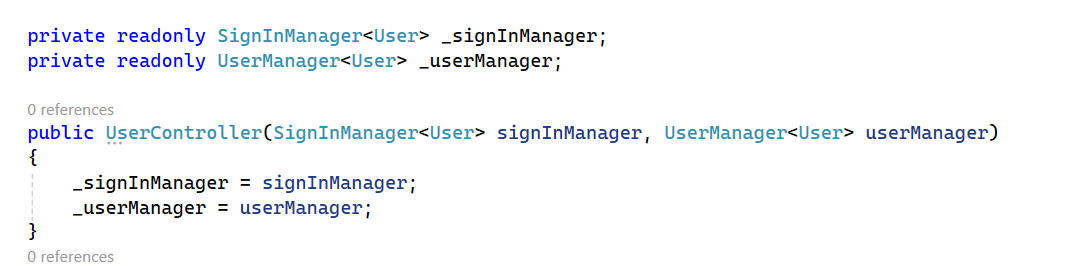
کلاسUser (Entity Model)

این کلاس نمایانگر موجودیت اصلی کاربر در پایگاه داده است و از کلاس پایه IdentityUser از فریم‌ورک ASP.NET Identity ارث‌بری می‌کند.شامل مشخصات اضافی مانند Name, Lname و ارتباط با سایر موجودیت‌ها Borrowings است.مدیریت کاربران در پایگاه داده از طریق این کلاس انجام می‌شود.هدف استفاده از آن نگهداری و مدیریت اطلاعات کاربران در پایگاه داده و پیاده‌سازی روابط با سایر جداول مثل ارتباط با امانت از طریق Borrowings

استفاده از DTO از ارسال مستقیم موجودیت‌های پایگاه داده به سمت کلاینت جلوگیری می‌کند و به افزایش امنیت و جداسازی منطق کمک می‌کند.داده‌های دریافتی از طریق UserDTO معمولاً پس از اعتبارسنجی، به مدل User تبدیل و در پایگاه داده ذخیره می‌شوند. UserDTO برای ارتباط با کاربر و اعتبارسنجی داده‌های ورودی استفاده می‌شود و User برای مدیریت داده‌های پایگاه داده و پیاده‌سازی منطق دامنه کاربرد دارد.

💡) نام های بکار برده در User و UserDTO باید باهم تفاوت داشته باشد. برای مثال Name و Lname

1. کنترلری میسازیم برای کاربر که دارای اکشن متد ثبت نام باشد. این اکشن متد هم صفحه ثبت نام را نمایش میدهد و هم به عمل ارسال اطلاعات به پایگاه داده می پردازد.

 Sign in man5ager و UserManagerرا در ابتدا کنترلر به صورت زیر inject می کنیم:

متغیر userManager یک نمونه از کلاس UserManager<Tuser> است که در ASP.NET Identity برای مدیریت عملیات کاربران استفاده می‌شود و وظایف این کلاس شامل ایجاد، حذف، به‌روزرسانی و مدیریت کاربران و رمزهای عبور آن‌ها است.

وقتی از سرویس Identity استفاده می کنیم باید ان را در program.cs معرفی کنیم:

builder.Services.AddIdentity<User, IdentityRole>().AddEntityFrameworkStores<ApplicationDbContext>().AddDefaultTokenProviders();

1. تابع Register را برای ثبت نام کاربر ایجاد میکنیم.



این کد مربوط به یک کنترلر در ASP.NET Core است که عملیات **ثبت‌نام کاربر (Register)** را مدیریت می‌کند. در ادامه، عملکرد هر بخش به طور خلاصه توضیح داده شده است:

متد GET

public IactionResult Register()

{

return View();

}

وقتی کاربر به صفحه ثبت‌نام دسترسی پیدا می‌کند، این متد یک View (صفحه ثبت‌نام) را نمایش می‌دهد.

متد POST

[HttpPost]

public IActionResult Register(UserDTO register)

این متد برای پردازش داده‌های فرم ثبت‌نام ارسال‌شده توسط کاربر استفاده می‌شود.

ورودی این متد یک مدل به نام UserDTO است که شامل اطلاعات ارسال‌شده مثل Username و Email است.

بررسی اعتبار مدل

if (ModelState.IsValid != true)

{

return View(register);

}

بررسی می‌کند که داده‌های ارسال‌شده مطابق قوانین اعتبارسنجی مدل هستند یا خیر. اگر نامعتبر باشند، همان صفحه ثبت‌نام را با داده‌های ارسال‌شده دوباره نمایش می‌دهد (برای جلوگیری از حذف اطلاعات کاربر).

ایجاد شیء کاربر جدید

var newUser = new User

{

UserName = register. Username,

Email = register. Email,

Name = register.FirstName,

Lname = register.LastName,

};

یک شیء جدید از کلاس User ساخته می‌شود که اطلاعات کاربر را از مدل UserDTO دریافت می‌کند.

توجه: در این مرحله هنوز این کاربر در پایگاه داده ذخیره نشده است.

این بخش از کد مربوط به بررسی تطابق رمز عبور و تأیید رمز عبور در فرآیند ثبت‌نام است.

If (register.Password != register.ConfPassword)

{

ModelState.AddModelError(“”, “رمز عبور و تایید رمز عبور با هم تطابق ندارند.”);

return View(register);}

شرط بررسی تطابق رمز عبور:

if (register.Password != register.ConfPassword)

این شرط بررسی می‌کند که مقدار Password و ConfPassword (تأیید رمز عبور) که کاربر وارد کرده است، برابر هستند یا خیر.اگر برابر نباشند، کاربر نمی‌تواند ادامه دهد، زیرا وارد کردن رمز عبور و تأیید آن به صورت یکسان برای اطمینان از صحت اطلاعات ضروری است.

افزودن خطا به ModelState

ModelState.AddModelError(“”, “ رمز عبور و تایید رمز عبور با هم تطابق ندارند”);

اگر شرط بالا برقرار باشد (یعنی رمزها برابر نباشند)، این خطا به شیء ModelState اضافه می‌شود.

پیام خطا ("رمز عبور و تایید رمز عبور با هم تطابق ندارند.") در نمای (View) به کاربر نشان داده خواهد شد.

آرگومان اول ("") تعیین می‌کند که خطا به یک فیلد خاص مرتبط نیست و به صورت عمومی نمایش داده می‌شود.

بازگشت به نمای ثبت‌نام:

return View(register);

پس از ثبت خطا، کد با استفاده از return View(register) کاربر را دوباره به صفحه ثبت‌نام هدایت می‌کند.

مقادیر وارد شده در فرم (مثل Username، Email) حفظ می‌شوند تا کاربر مجبور به وارد کردن دوباره آن‌ها نباشد.

این خط از کد مسئول ذخیره‌سازی کاربر جدید در سیستم است و از UserManager یک سرویس مدیریت کاربران در ASP.NET Identity استفاده می‌کند.

var result = \_userManager.CreateAsync(newUser, register.Password).Result;

متد CreateAsync یک کاربر جدید را در پایگاه داده ایجاد می‌کند. رمز عبور به صورت خودکار هش (Hash) شده و با یک Salt در پایگاه داده ذخیره می‌شود تا امنیت افزایش یابد.

سپس کد های زیر را داریم:

if (result.Succeeded)

{

return RedirectToAction(“Login”, “Account”);

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(“”, error.Description);

}

این بخش از کد مربوط به بررسی نتیجه عملیات ایجاد کاربر از متد CreateAsync است و به ترتیب موفقیت یا شکست عملیات واکنش نشان می‌دهد. بیایید گام به گام توضیح دهیم:

بررسی موفقیت عملیات:

if (result.Succeeded)

{

return RedirectToAction(“Login”, “Account”);

}

result.Succeeded : اگر مقدار true باشد، نشان می‌دهد که عملیات ایجاد کاربر با متد CreateAsync با موفقیت انجام شده است.

RedirectToAction(“Login”, “Account”):

کاربر را به اکشن Login در کنترلر Account هدایت می‌کند.

این معمولاً به این معناست که ثبت‌نام کامل شده و کاربر باید وارد صفحه ورود شود تا وارد سیستم شود.

مدیریت خطاها در صورت شکست عملیات:

foreach (var error in result.Errors)

{

ModelState.AddModelError(“”, error.Description);

}

result.Errors:

اگر عملیات شکست بخورد، این خاصیت شامل لیستی از خطاها خواهد بود.

هر خطا از نوع IdentityError است و یک Description (توضیح خطا) دارد.

ModelState.AddModelError(“”, error.Description):

هر خطا به ModelState اضافه می‌شود تا در نمای (View) نمایش داده شود.

آرگومان اول: ("") نشان می‌دهد که خطا به یک فیلد خاص مرتبط نیست و به صورت عمومی نمایش داده می‌شود.

آرگومان دوم: پیام خطا (مانند "ایمیل قبلاً استفاده شده است" یا "رمز عبور خیلی ضعیف است").

بازگشت به نمای ثبت‌نام:

return View(register);

اگر عملیات شکست بخورد و خطاها اضافه شوند، کاربر به صفحه ثبت‌نام بازمی‌گردد.

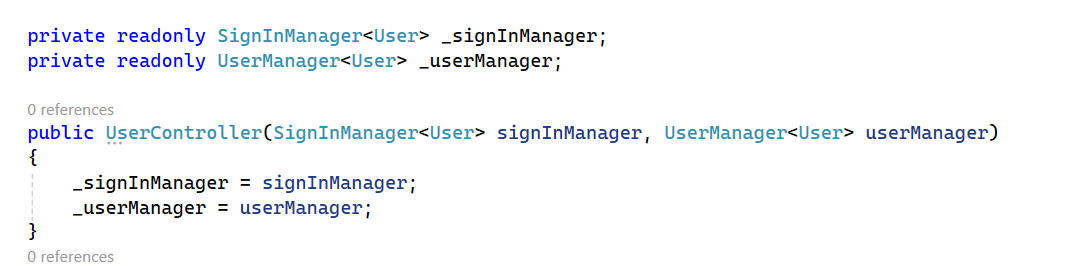
اطلاعات قبلی وارد شده مثل Email و Username در فرم باقی می‌مانند، زیرا register به ویو فرستاده می‌شود.

نام فایل: Accounting

**Sign in Manager**

تا به اینجا توسط user management یک user با مشخصاتی همچون نام و نام خانوادگی و ایمیل ایجاد کرده ایم. حالا می خواهیم توسط sign in manager عمل sign in را پیاده سازی کنیم.

در مراحل قبل ما مدلی به نام user ساختیم که از Identity User ارث بری می کند و کنترلری ساختیم و در آن یک نمونه از User Manager ساختیم:



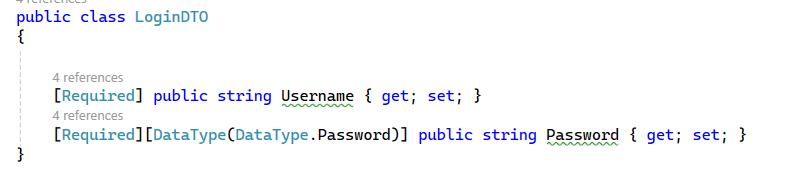
سپس مدلی ساختیم به نام UserDTO که شامل نام کاربری و رمز و غیره میشود و یک view ساختیم که در صورت وارد کردن اطلاعات به فرم ، اطلاعات را به اکشن Register میفرستد.

سپس توسط Model State بررسی می کنیم که آیا اطلاعات به درستی وارد شده؟ اگر درست بود کاربری با مشخصاتی که در فرم بود ایجاد شود.

سرویس User Manager دارای متدی به نام create async است که توسط آن میتوانیم یک کاربر با نام کاربری و رمز Hash شده ایجاد کنیم.

تا به اینجا مراحل ثبت نام کاربر به پایان رسید. برای پیاده سازی Login به سرویس Signin Manager نیاز داریم.

ابتدا یک مدل برای رد و بدل کردن اطلاعات لازم داریم به نام LoginDTO :



سپس در کنترلر یک اکشن متد برای Login مینویسیم:



**3**

1

**2**

1

**1**

1

ابتدا یک متد از جنس Get برای نمایش صفحه لاگین و سپس متدی از نوع Post را ایجاد می کنیم.

[HttpGet]

public IActionResult Login()

{

return View();

}

اکشن متد Login به عنوان ورودی LoginDTO را دریافت میکند.

این متد دارای یک view است که در آن فرم لاگین قرار می گیرد و نام کاربری و پسورد از کاربر دریافت می شود. درصورتی که مقادیر وارد شده درست باشد دستورات مربوط به لاگین انجام می شود.

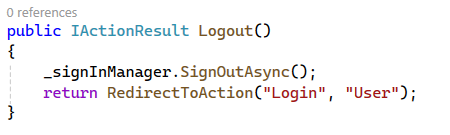
1. سرویس User Manager دارای متدی به نام FindByNameAsync است که میتواند دنبال نام کاربری در جدول LoginDTO بگردد.

💡) چون تابع از نوع async است در نهایت خروجی را به Result تبدیل میکنیم.

اگر کاربر وجود نداشت ارری با عنوان نام کاربری وجود ندارد نمایش میدهیم.

1. تابع PasswordSignInAsync دارای چند ورودی است: ورودی اول user که با FindByNameAsync آن را پیدا کردیم، ورودی دوم پسورد و ورودی سوم و چهارم دو متغیر bool اند که اولی برای تیک گزینه remember است و دومی برای جلوگیری از چندین ورود ناموفق.
2. اگر Result موفق بود کاربر میتواند به صفحه اصلی index در کنترلر Home دسترسی پیدا کند. و در غیر این صورت پیغام خطایی برای غلط بودن رمز عبور به کاربر نمایش میدهیم.

برای پیاده سازی Logout یک اکشن متد می سازیم و توسط تابع Signout Async عمل خروج انجام میشود.



? میخواهیم وقتی کاربر Login کرد دکمه Logout در کنار صفحه نمایش پیدا کند:

برای انجام این کار به \_Layout.cs رفته و دستور زیر را در تگ <Li> قرار میدهیم.

@if (User.Identity.IsAuthenticated)

{

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-controller**="User"

**asp-action**="Logout">Logout</**a**>

</li>

}

نام فایل: Accounting

تمرین) کتابخانه ای را طراحی کنید که کاربر فقط با ثبت نام بتواند کتاب قرض بگیرد.

چالش های برنامه:

* صفحه لاگین و ثبت نام
* وضعیت قرض گرفته شدن را نشان دهید.
* از اینترفیس استفاده شود.

نام فایل: LibraryWinterface

در مرحله اول برنامه را پوشه بندی می کنیم : در پوشه model پوشه ای به نام Entity برای موجودیت هایی مثل کتاب و کاربر ایجاد می کنیم. و همچنین برای DTO ها نیز به پوشه ای نیاز داریم . میتوانید پوشه ای هم برای کلاس های کمکی برنامه داشته باشید که ما در اینجا نام این پوشه را Util قرار داده ایم.

ارتباطات در پایگاه داده ما به صورت زیر است:

سه موجودیت با نام هایAppuser و Book و Request داریم که جدول درخواست ها واسطه ای بین کاربر و کتاب است.

Book

Request

User

AppUser ID

Book ID

مدل Book

public class Book

{

public string Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public string Author { get; set; }

public string Status { get; set; }

public DateTime CreatedAt { get; set; }

}

مدل Request

public class Request

{

public string Id { get; set; }

//navigation : User

public string AppUserId { get; set; }

public virtual AppUser AppUser { get; set; }

//navigation : Book

public string BookId { get; set; }

public virtual Book Book { get; set; }

public string RequestStatus { get; set; }1

public DateTime RequestDate { get; set; }2

}

جدول درخواست توسط ID کاربر و کتاب با آنها در ارتباط است. و از طریق nav prop میتوان به جداول دیگر دسترسی پیدا کرد.

1. درخواست ها باید وضعیت و تاریخ داشته باشند.

مدل AppUser

public class AppUser : IdentityUser

{

public string? FirstName { get; set; }

public string? LastName { get; set; }

public virtual List<Request> Requests { get; set; }**1**

{

1. هر کاربر می تواند لیستی از درخواست ها داشته باشد.
2. **؟** به این معناست که فیلد میتواند خالی باشد و پر کردن آن اختیاری است.

در پوشه Data ما کلاسی به نام ApplicationDbContext می سازیم که بین جدول ها ارتباط برقرار می کند . این کلاس باید از IdentityDbContext ارث بری کند و کتابخانه آن به پروژه اضافه شود.

Using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : IdentityDbContext<AppUser>

{

public ApplicationDbContext(DbContextOptions options) : base(options)

{

}

public DbSet<Book> Books { get; set; }

public DbSet<Request> Requests { get; set; }

}

< > به معنای genericبودن است و نوع را مشخص میکند. ApplicationDbContext از IdentityDbContext ارث بری میکند و نوع آن از جنس AppUser است.

جدول درخواست و کتاب را DbSet می کنیم ولی جدول چون ApplicationDbContext از جدول کاربر ارث بری می کند نیازی به DbSet کردن این جدول نیست.

به program.cs رفته و دستورات زیر را به آن اضافه کنید:

// DbContext

builder.Services.AddDbContext<ApplicationDbContext>(option =>

{

option.UseSqlServer(builder.Configuration.GetConnectionString(“DefaultConnection”));

});

// Identity

builder.Services.AddIdentity<AppUser, IdentityRole>(option =>

{

option.Password.RequireNonAlphanumeric = false;

option.Password.RequiredLength = 8;

option.Password.RequireUppercase = false;

option.Password.RequireLowercase = false;

option.User.RequireUniqueEmail = false;

option.SignIn.RequireConfirmedPhoneNumber = false;

option.SignIn.RequireConfirmedEmail = false;

option.SignIn.RequireConfirmedPhoneNumber = false;

})

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

در ادامه ما نیاز به توابعی داریم تا عمل ثبت نام ورود و خروج کاربر را پیاده سازی کنند. هر کدام از این تابع ها ورودی های متفاوتی دارند.

سرویس UserManager دارای فیلد های زیادی است که بسته به نوع سایت از فیلد های آن استفاده می کنیم. برای مثال برخی از سرویس ها برای خدمات دهی فقط نیاز به نام و شماره تلفن دارند.

برای اینکه فقط فیلد های مورد نظر را از کاربر دریافت کنیم کلاسی میسازیم که جدول نیست و در پایگاه داده وجود ندارد. فقط کلاسی است برای دریافت اطلاعات که به این کلاس ها Data Transfer Object یا به اختصار DTO می گویند.

Data Transfer Object (DTO) یک شیء ساده است که به طور معمول شامل داده‌ها است و هیچ منطقی (Business Logic) در آن وجود ندارد. هدف اصلی DTO این است که داده‌ها را از یک سیستم به سیستم دیگر منتقل کند، به ویژه زمانی که داده‌ها باید از طریق شبکه یا بین لایه‌های مختلف یک برنامه منتقل شوند.

DTO معمولاً فقط حاوی داده‌ها است و هیچ متد پیچیده یا رفتار خاصی ندارد . استفاده از DTO می‌تواند به کاهش تعداد درخواست‌های شبکه‌ای یا تعاملات پایگاه داده کمک کن.

* کاهش وابستگی‌ها: در بسیاری از موارد، DTO به کاهش وابستگی به مدل‌های پایگاه داده یا شیءهای دیگر کمک می‌کند، به طوری که این مدل‌ها فقط برای انتقال داده استفاده می‌شوند.

مثال:فرض کنید که یک برنامه وب در حال تعامل با یک پایگاه داده است و اطلاعات مربوط به کاربران را از آن دریافت می‌کند. به جای اینکه کل شیء مدل پایگاه داده را به عنوان نتیجه API بازگشت دهد (که می‌تواند شامل داده‌های اضافی یا منطقی باشد که نیازی به ارسال آن‌ها نیست)، می‌توان از یک DTO استفاده کرد.

در این برنامه ما از یک DTO برای دریافت اطلاعات ثبت نام استفاده می کنیم. در پوشه model یک پوشه DTO میسازیم و یک کلاس به نام registerDTO به آن اضافه میکنیم:

public class RegisterDTO

{

[Required]

public string UserName { get; set; }

[EmailAddress]1

public string Email { get; set; }

[Required]

[DataType(DataType.Password)]2

public string Password { get; set; }

[Required, DataType(DataType.Password)]

[Compare(nameof(Password))]3

[Display(Name = “Confirme password”)]4

public string ConfirmePassword { get; set; }

}

1. ویژگی از نوع و با فرمت ایمیل دریافت شود.
2. نوع داده دریافتی از جنس پسورد است.
3. این ویژگی برای مقایسه دو فیلد در مدل استفاده می‌شود و معمولاً برای تأیید مطابقت رمز عبور و تأیید رمز عبور (Confirm Password) به کار می‌رود.
4. این ویژگی برای تعیین نام نمایشی یک فیلد در رابط کاربری استفاده می‌شود و در اینجا نام "Confirme password" برای فیلد مربوطه نمایش داده می‌شود.

همچنین برای صفحه لاگین نیز به یک DTO نیاز داریم:

public class LoginDTO

{

[Required]

public string UserName { get; set; }

[Required]

[DataType(DataType.Password)]

public string Password { get; set; }

[Display(Name = “Remember Me?”)]1

public bool RememberMe { get; set; }

}

1. این ویژگی به شما این امکان را می‌دهد که نام فیلد را در رابط کاربری به صورت دلخواه و قابل فهم برای کاربر نمایش دهید. به جای اینکه نام فیلد مستقیماً از نام متغیر کد منبع گرفته شود، می‌توانید یک متن قابل فهم‌تر و دوستانه‌تر برای کاربر ارائه دهید. در این مورد خاص، "Remember Me" یک گزینه است که معمولاً در فرم‌های ورود (Login Forms) وجود دارد. این گزینه به کاربر اجازه می‌دهد که انتخاب کند آیا می‌خواهد سیستم او را به خاطر بسپارد یا خیر، به طوری که در بازدیدهای بعدی نیازی به ورود مجدد اطلاعات نداشته باشد.

برای پیاده سازی لاگین و ثبت نام نیاز به متد هایی داریم. متد ها را میتوان به صورت مستقیم در کنترلر نوشت و یا توسط یک اینترفیس یک مجموعه ای از قوانین ایجاد کرد.

پوشه ای برای اینترفیس ها ایجاد کنید و اینترفیسی به نام IaccountService را به آن اضافه کنید.

💡) اینترفیس ها باید با I شروع شوند.

Public interface IaccountService

{

Task<GeneralServiceRespone> RegisterAsync(RegisterDTO registerDto);

Task<GeneralServiceRespone> LoginAsync(LoginDTO oginDto);

Task<ServiceResponse<AppUser>> GetUserInfoAsync(ClaimsPrincipal user);

Task LogoutAsync();}

ساختار اینترفیس همانند متدی است که درون آن دستوری قرار ندارد. تابع ها میتوانند خروجی هایی از نوع Async و Await داشته باشند. یعنی تابع اول صبر میکند تا نتیجه درخواست را دریافت و سپس سراغ مرحله بعد برود. زیرا هنگامی که درخواست به سرور ارسال می شود تاخیری در جواب دهی وجود دارد. برای استفاده از این نوع تابع از کلمه کلیدی Task استفاده می شود.

* برای پیاده سازی ثبت نام تابعی میسازیم که در ورودی کلاس مدل را دریافت می کند و سپس با مکث کلاسی به نام GeneralServiceRespone را به عنوان خروجی نشان میدهد که جلوتر جزئیات کلاس را خواهیم دید:

Task<GeneralServiceRespone>**1** RegisterAsync(RegisterDTO registerDto);

1. خروجی این تابع از جنس کلاس GeneralServiceRespone است.

کلاس GeneralServiceRespone را در پوشه مدل و زیر پوشه Util ایجاد کنید:

public class GeneralServiceRespone

{

public bool IsSuccess { get; set; }1

public string Message { get; set; }

public List<string> Errors { get; set; }

}

1. ثبت نام موفقیت آمیز بوده یا نه؟

خروجی متد ثبت نام یک bool که نشان دهنده موفقیت آمیز بودن عملیات است و پیغام خطا و لیستی از ارور هاست.

* متد لاگین هم همانند ثبت نام نوشته میشود و خروجیش GeneralServiceRespone کلاس است.

Task<GeneralServiceRespone> LoginAsync(LoginDTO oginDto);

* متد بعدی GetUserInfoAsync است که اطلاعات کاربر را دریافت میکند و نتایجی را ارسال میکند.

Task<ServiceResponse<AppUser>> GetUserInfoAsync(ClaimsPrincipal user);

ورودی تابع GetUserInfoAsync عبارت ClaimsPrincipal user است.

**توضیحات:**

**ClaimsPrincipal user** این پارامتر یک شیء از نوع ClaimsPrincipal است که به طور معمول در سیستم‌های احراز هویت برای نمایندگی اطلاعات مربوط به کاربر استفاده می‌شود.

ClaimsPrincipal نماینده‌ی یک کاربر است و اطلاعات مختلفی درباره‌ی کاربر مانند **نام کاربری**، **ایمیل**، **نقش‌ها** و دیگر اطلاعات احراز هویت یا **ادعاها (Claims)** را در خود دارد.

این شیء معمولاً از پس از عملیات ورود (Authentication) توسط سیستم احراز هویت مثل ASP.NET Core Identity یا Oauth به دست می‌آید و می‌تواند شامل اطلاعات کاربری مانند نام، ایمیل، نقش‌ها و غیره باشد.

**نحوه استفاده:** تابع GetUserInfoAsync با استفاده از پارامتر ClaimsPrincipal user می‌تواند اطلاعات کاربری که در حال حاضر وارد شده است را پردازش کند. این شیء به شما امکان می‌دهد که از **ادعاها (Claims)** برای دسترسی به ویژگی‌های مختلف کاربر (مثل شناسه کاربری یا نقش‌ها) استفاده کنید.

**مثال:** در یک پروژه ASP.NET Core، معمولاً ClaimsPrincipal از طریق احراز هویت و از طریق **سیستم هویت** مانند JWT یا کوکی‌ها به دست می‌آید. در این صورت، ClaimsPrincipal می‌تواند حاوی اطلاعاتی مانند شناسه کاربری (user id)، نقش‌ها، ایمیل و غیره باشد.

پس میتوان گفت متدی که ما ساختیم با نام GetUserInfoAsync به عنوان ورودی یک آبجکت را دریافت می کند که اطلاعات کاربر را در خود دارد. و خروجی، کلاس کمکی ای است به نام ServiceResponse که از جنس AppUser است.

* کلاس ServiceResponse را در پوشه مدل و زیر پوشه Util ایجاد کنید:

public class ServiceResponse<T>

{

public bool IsSuccess { get; set; }

public T? Data { get; set; }

public string? ErrorMessage { get; set; }

public static ServiceResponse<T> Success(T data)

{

return new ServiceResponse<T> { IsSuccess = true, Data = data };

}

public static ServiceResponse<T> Failure(string errorMessage = “”)

{

return new ServiceResponse<T> { IsSuccess = false, ErrorMessage = errorMessage }; }}

در اینجا، T یک **نوع جنریک** است که می‌تواند هر نوع داده‌ای (مثل int، string، یا هر کلاس دلخواه) باشد. این به شما این امکان را می‌دهد که کلاس ServiceResponse را برای هر نوع داده‌ای (مثلاً نتیجه یک عملیات یا اطلاعات برگشتی) استفاده کنید.

* ServiceResponse<int> برای بازگشت عدد.
* ServiceResponse<AppUser> برای بازگشت اطلاعات یک کاربر.
* و در آخر تابعی بدون خروجی برای logout مینویسیم:

Task LogoutAsync();

تا به اینجا اینترفیس را ایجاد کردیم. اما هر اینترفیس نیاز به یک کلاس برای پیاده سازی قوانین دارد. پوشه ای به نام Services بسازید تا کلاس های پیاده سازی را در آن قرار دهیم. این کلاس ها هم میتوانند در اینترفیس تعریف شوند و هم به طور جداگانه نوشته شوند.

* کلاسی به نام AccountService در این پوشه ایجاد کنید که از اینترفیس IaccountService ارث بری کند:

با زدن ctrl+ . میتوانید تمام قوتنین را پیاده سازی کنید:

public class AccountService : IaccountService

{

public Task<ServiceResponse<AppUser>> GetUserInfoAsync(ClaimsPrincipal user)

{

throw new NotImplementedException();

}

public Task<GeneralServiceRespone> LoginAsync(LoginDTO oginDto)

{

throw new NotImplementedException();

}

public Task LogoutAsync()

{

throw new NotImplementedException();

}

public Task<GeneralServiceRespone> RegisterAsync(RegisterDTO registerDto)

{

throw new NotImplementedException();

}

}

سپس باید نمونه ای از سرویس های زیر بسازیم.

این کد سازنده‌ای برای کلاس AccountService است که وابستگی‌ها مانند SignInManager و UserManager و Ilogger را از طریق Dependency Injection دریافت و در فیلدهای خصوصی ذخیره می‌کند.

Private readonly SignInManager<AppUser> \_signInManager;

private readonly UserManager<AppUser> \_userManager;

private readonly Ilogger<AccountService> \_logger;

public AccountService(SignInManager<AppUser> oginDtoager, UserManager<AppUser> userManager, Ilogger<AccountService> logger)

{

\_signInManager = oginDtoager;

\_userManager = userManager;

\_logger = logger; }

حالا به تعریف ویژگی های تابع می پردازیم:

**تابع RegisterAsync**

این تابع برای ثبت‌نام کاربر جدید طراحی شده است. ابتدا بررسی می‌کند که آیا نام کاربری وارد شده قبلاً وجود دارد یا خیر. اگر نام کاربری موجود باشد، یک پاسخ با خطای “Username already taken” برمی‌گرداند. در غیر این صورت، کاربر جدید ساخته می‌شود و تلاش می‌کند تا کاربر را در پایگاه داده ذخیره کند. اگر ثبت‌نام موفق باشد، پیام “Registration successful” برمی‌گرداند. در صورت بروز خطا، پیغام‌های خطا را جمع‌آوری کرده و به کاربر ارسال می‌کند.

Public async Task<GeneralServiceRespone> RegisterAsync(RegisterDto registerDto)

{

var isUserExists = await \_userManager.FindByNameAsync(registerDto.UserName);

if (isUserExists != null) {

return new GeneralServiceRespone()

{

IsSuccess = false,

Message = “Registration failed.”,

Errors = new List<string>() { “Username already taken.” }

};

}

AppUser newUser = new AppUser()

{

UserName = registerDto.UserName,

Email = registerDto.Email,

};

var result = await \_userManager.CreateAsync(newUser, registerDto.Password);

if (result.Succeeded)

{

return new GeneralServiceRespone()

{

IsSuccess = true,

Message = “Registration successful.”

};

}

var errors = new List<string>();

foreach (var error in result.Errors)

{

errors.Add(error.Description);

}

return new GeneralServiceRespone()

{

IsSuccess = false,

Message = “”,

Errors = errors

};

}

**تابع LoginAsync**

این تابع برای ورود کاربران استفاده می‌شود. با استفاده از اطلاعات ورود (نام کاربری و رمز عبور)، سعی می‌کند کاربر را شناسایی کند. در صورت موفقیت، پیام “Login was successful” را برمی‌گرداند. در غیر این صورت، پیامی شامل خطای “Username or password is wrong” ارسال می‌کند.

Public async Task<GeneralServiceRespone> LoginAsync(LoginDto oginDto)

{

var result = await \_signInManager.PasswordSignInAsync(oginDto.UserName, oginDto.Password, oginDto.RememberMe, true);

if (result.Succeeded)

{

return new GeneralServiceRespone()

{

IsSuccess = true,

Message = “Login was successful.”

};

}

return new GeneralServiceRespone()

{

IsSuccess = false,

Message = “Login failed.”,

Errors = new List<string>() { “Username or password is wrong, try again!” }

};

**تابع LogoutAsync**

این تابع برای خروج از سیستم استفاده می‌شود. با استفاده از متد SignOutAsync، کاربر را از سیستم خارج می‌کند.

Public async Task LogoutAsync()

{

await \_signInManager.SignOutAsync();

}

**تابع GetUserInfoAsync**

این تابع اطلاعات کاربر وارد شده را می‌گیرد. ابتدا تلاش می‌کند که کاربر را پیدا کند. اگر کاربر یافت نشود، پیامی با خطای “No user found” برمی‌گرداند. اگر کاربر یافت شود، اطلاعات وی را به صورت موفقیت‌آمیز باز می‌گرداند. در صورت بروز خطا، خطا را لاگ کرده و پیغام “Database error” را باز می‌گرداند.

Public async Task<ServiceResponse<AppUser>> GetUserInfoAsync(ClaimsPrincipal user)

{

var userInfo = await \_userManager.GetUserAsync(user);

if (userInfo == null)

{

return ServiceResponse<AppUser>.Failure(“No user found.”);

}

try

{

return ServiceResponse<AppUser>.Success(userInfo);

}

catch (Exception ex)

{

\_logger.LogError(ex, “An error occurred while getting user info.”);

return ServiceResponse<AppUser>.Failure(“Database error”);

throw;

}

}

کنترلر Account

* این توابع باید در کنترلر صدا زده شوند و صفحات مخصوصی داشته باشند پس کنترلری به نام AccountController میسازیم:

private readonly IaccountService \_authService;

public AccountController(IaccountService authService)

{

\_authService = authService;

}

این کد وابستگی IaccountService را از طریق Dependency Injection به کنترلر AccountController تزریق می‌کند. سازنده کلاس، نمونه IaccountService را دریافت کرده و به فیلد \_authService اختصاص می‌دهد تا بتوان از آن برای انجام عملیات‌های مرتبط با حساب کاربری استفاده کرد.

**تابع Register (GET)** این تابع فقط نمایش فرم ثبت‌نام را انجام می‌دهد. هیچ عملیات خاصی انجام نمی‌دهد و تنها ویو ثبت‌نام را باز می‌کند.

Public IactionResult Register()

{

return View();}

**تابع Register (POST)**

این تابع درخواست ثبت‌نام را پردازش می‌کند. ابتدا بررسی می‌کند که آیا داده‌های فرم معتبر هستند یا نه. اگر داده‌ها معتبر نباشند، فرم با ارورها به کاربر باز می‌گردد. در غیر این صورت، تلاش می‌کند که کاربر را از طریق سرویس ثبت‌نام کند. اگر ثبت‌نام موفق باشد، کاربر به صفحه ورود هدایت می‌شود. در غیر این صورت، ارورهای مربوطه به مدل افزوده می‌شود و فرم دوباره نمایش داده می‌شود.

[HttpPost]

public async Task<IactionResult> Register(RegisterDto registerDto)

{

if (!ModelState.IsValid)

{

return View(registerDto);

}

var result = await \_authService.RegisterAsync(registerDto);

if (result.IsSuccess)

{

return RedirectToAction(“Login”);

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(“”, error);

}

return View(registerDto);

}

**تابع Login (GET)**

این تابع فرم ورود را نمایش می‌دهد و هیچ عملیاتی انجام نمی‌دهد. فقط ویو ورود را برمی‌گرداند.

Public IactionResult Login()

{

return View();

}

**تابع Login (POST)**

این تابع درخواست ورود کاربر را پردازش می‌کند. ابتدا بررسی می‌کند که آیا داده‌های فرم ورود معتبر هستند یا نه. اگر داده‌ها معتبر نباشند، فرم با ارورها به کاربر باز می‌گردد. در صورت معتبر بودن، تلاش می‌کند که کاربر را وارد سیستم کند. اگر ورود موفق باشد، کاربر به صفحه اصلی هدایت می‌شود. در غیر این صورت، ارورهای مربوطه به مدل افزوده می‌شود و فرم دوباره نمایش داده می‌شود.

[HttpPost]

public async Task<IactionResult> Login(LoginDto oginDto)

{

if (!ModelState.IsValid) return View(oginDto);

var result = await \_authService.LoginAsync(oginDto);

if (result.IsSuccess) return RedirectToAction(“Index”, “Home”);

foreach (var error in result.Errors)

{

ModelState.AddModelError(“”, error);

}

return View(oginDto);

}

**تابع Logout**

این تابع کاربر را از سیستم خارج می‌کند. پس از خروج، کاربر به صفحه ورود هدایت می‌شود.

[HttpPost]

[ValidateAntiForgeryToken]1

public async Task<IactionResult> Logout()

{

await \_authService.LogoutAsync();

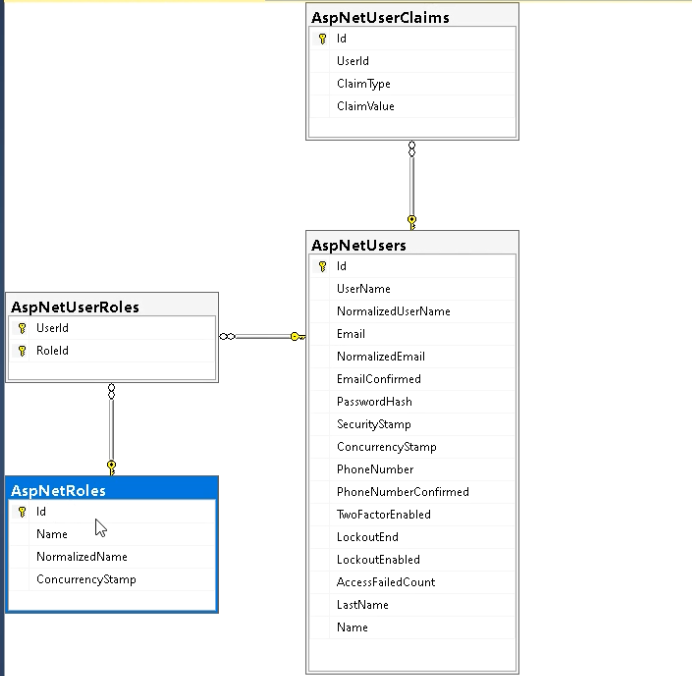
return RedirectToAction(“Login”);}

* 1. ازین دستور برای مسائل امنیتی و جلوگیری از اینکه فرد دیگری logout کند استفاده میشود.

نام فایل: Accounting

**بخش هشتم (Role)**

کاربر ها در سایت باید دارای سطح دسترسی باشند. برای پیاده سازی این عمل از جداول کتابخانه Identity استفاده می کنیم.



جدول role با جدول user از طریق یک جدول واسط به نام user role در ارتباط است.

در مرحله اول در program.cs برای احراز هویت یک builder می گذاریم:

builder.Services.AddAuthentication();

builder.Services.AddIdentity<User, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders().AddRoles<IdentityRole>()

.AddErrorDescriber<IdentityErrorDescriber>();

builder.Services.AddAuthorization(option =>

{

option.AddPolicy("Test",policy =>

{

policy.RequireClaim("testest", "itis");

});

}); //new

کنترلری برای Role می سازیم و چون مشخص کردن سطح دسترسی به عهده ادمین است با یک Area دستورات مربوط به ادمین را از کاربر عادی جدا میکنیم. برای ساخت Area مسیر زیر را بروید:

Add> new scaffold > new area > name

💡) همه پکیج ها را به اخرین نسخه آپدیت کنید تا به خطا بر نخورید.

سپس یک map controller routeدر program.cs ایجاد کنید:

app.MapControllerRoute(

name: "areas",

pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(

name: "default",

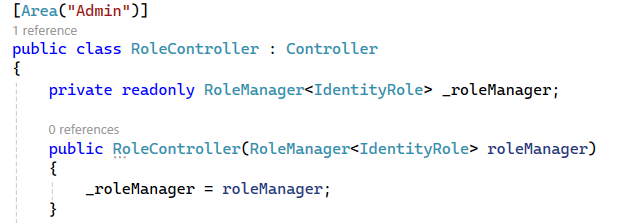
pattern: "{controller=Home}/{action=Index}/{id?}");

نکته تکمیلی) کنترلرهایی که داخل Area هستند مثل UserController و RoleController در Areas/Admin/Controllers/ باید این اتریبیوت بالا داشته باشند:

[Area("Admin")]

public class UserController : Controller{…}

کنترلر Role را در Area ادمین ایجاد میکنیم و نام Area را بالای کنترلر می نویسیم.



در کنترلر Role توسط RoleManager میتوان به جدول Role در پکیج Identity دسترسی پیدا کرد و به Role ها اضافه یا حذف کرد.

جدول Role شامل ویژگی هایی میشد که اگر میخواهید به آن ها اضافه کنید (مثل name و Lname که با استفاده از مدل User اضافه کردیم ) میتوانید مدلی برای Role بسازید که از IdentityRole ارث بری کند و ویژگی های اضافه ای نیز داشته باشد با نام RoleDTO.

اول کلاسی به نام RoleDTO میسازیم که برای گرفتن نام نقش از کاربر در فرم‌ها استفاده می‌شه.  
یعنی فقط یه مدل ساده‌ست که توی فرم به‌کار می‌ره.

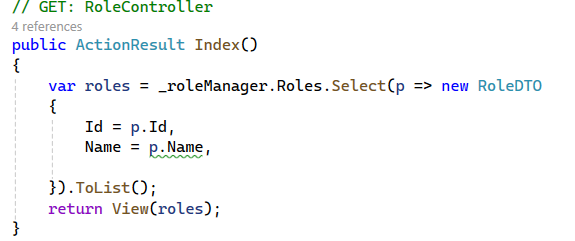
public class RoleDTO{

public string Name { get; set; }

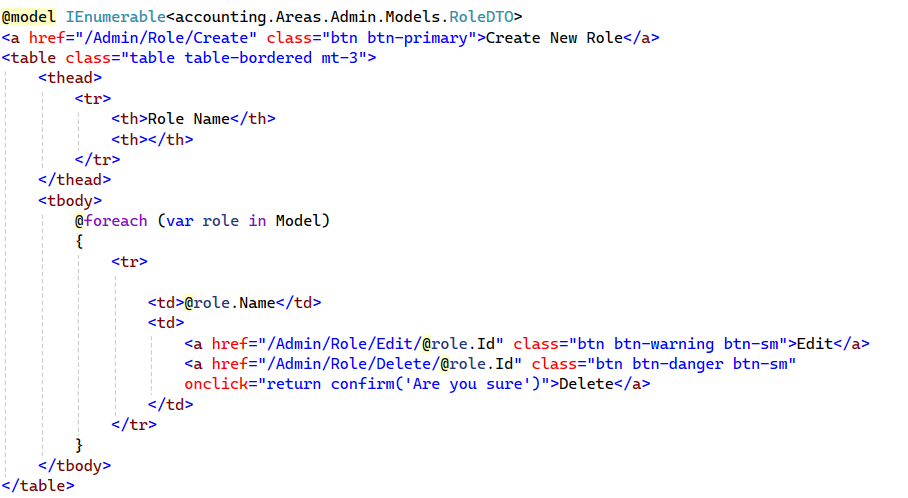
public string Id { get; set; } // فقط در صورت نیاز به ID}

💡) وقتی از پکیج identity استفاده می کنیم نوع id باید از جنس string باشد.

در قسمت اول میخواهیم در صفحه index این کنترلر لیستی از Role ها داشته باشیم:



توسط Select از Role ها Id و Name را نیاز داریم. آن ها را جدا می کنیم و لیستی میسازیم و آن را به ویو index پاس می دهیم.



💡) اگر در ویوی Razor، تگ‌هِلپرهایی مثل asp-action کار نمی‌کنندیعنی ویژوال استودیو کد تگ‌هلپرهای Razor را نمیشناسد. برای حل این مشکل فایل import view به پوشه Role کپی کنید. وقتی یک Area ساخته می شود باید فایل های مربوط به Razor که در پوشه Shared قراردارند در Area نیز وجود داشته باشند.

سپس متدی برای ایجاد نقش جدید (role) در سیستم Identity طراحی می کنیم. به‌عنوان مثال، نقش‌هایی مثل "admin"، "user" یا "manager" .

متد افزودن نقش به شرح زیر است:

**1**

1. ابتدا یک نمونه از Role می سازیم و سپس توسط CreateAsync نقش جدید را اضافه می کنیم و اگر عمل موفقیت آمیز بود به صفحه index میرویم.

در مرحله بعد view برای Create میسازیم:

@model accounting.Areas.Admin.Models.RoleDTO

<**form** **asp-action**="Create" method="post">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div >

<input type="submit" value="Create" class="btn btn-primary"/>

</div>

</**form**>

<div>

<a href="/Admin/Role" class="btn btn-secondary mt-2">Back To List</a>

</div>

?میخواهیم هرگاه یک کاربر جدید ایجاد شد به صورت پیش فرض کاربر عادی در نظر گرفته شود. به user controller میرویم و در متد ثبت نام دستور زیر را اضافه می کنیم:



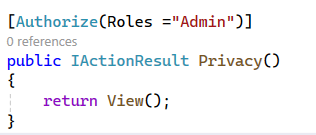
دستور AddToRoleAsync به جدول role کاربر را با سطح دسترسی عادی اضافه میکند. و سپس مقادیر در متغیر Result2 قرار می گیرد.

اگر ساخت و نقش دهی به کاربر به موفقیت انجام شد به اکشن index در کنترلر home می رویم.

? حالا اگر بخواهیم فقط ادمین بتواند وارد متدی خاص شود اول از همه باید یک role با نام ادمین ایجاد کنیم. سپس کاربری را به صورت دستی دارای نقش ادمین می کنیم.

با دستور زیر در کنترلر فقط افراد با همان سطح دسترسی می توانند محتوای متد را ببینند:

برای مثال فقط کاربر با دسترسی ادمین بتواند حتوای اکشن Privacy را ببیند:



**بخش نهم (Claim)**

تا به اینجا توسط Role به افراد سطح دسترسی دادیم. مفهوم دیگری وجود دارد به نام claim که توسط آن میتوان افرادی که لاگین می کنند را دسته بندی کنیم. برای مثال افراد زیر 18 سال نتوانند ورود کنند و یا برخی کاربران کاربر طلایی باشند و ...

Claim ها در سیستم کلاینت درون کوکی ها ذخیره می شوند و به محض ورود در لاگین کوکی هایی ساخته می شود.

در مرحله اول کنترلری برای Claimمی سازیم و چون به سرویس user Manager نیاز داریم CTOR برای آن میسازیم:

private readonly UserManager<User> \_userManager;

public ClaimController(UserManager<User> userManager)

{

\_userManager = userManager;

}

💡) ابتدای کنترلر [Area("Admin")] را قرار دهید.

در اکشن Index میتوانیم Claim هارا به نمایش دربیاوریم:

[Authorize(Roles = "Admin")]

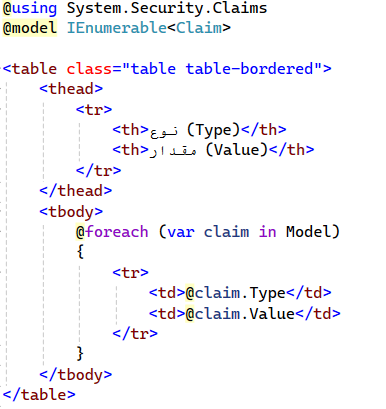
public IActionResult Index()

{

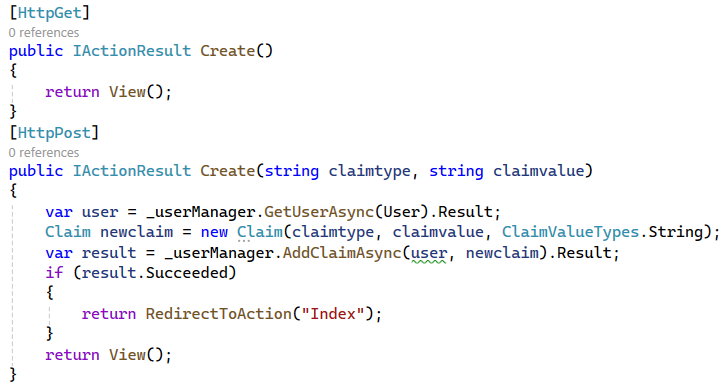
return View(User.Claims);

}

💡) اگر در ویوی Razor، تگ‌هِلپرهایی مثل asp-action کار نمی‌کنندیعنی ویژوال استودیو کد تگ‌هلپرهای Razor را نمیشناسد. برای حل این مشکل فایل import view به پوشه Claim کپی کنید.

توسط جدول User میتوانیم به Claims دست پیدا کنیم.

سپس متدی برای ایجاد Claim ها مینویسیم:



متد Create به‌طور خلاصه وظیفه‌ی افزودن یک Claim جدید به کاربر فعلی لاگین‌شده را دارد.   
متد دو پارامتر claimtype و claimvalue دریافت می‌کند که توسط کاربر در فرم وارد شده‌اند و با استفاده از \_userManager.GetUserAsync(User).Result، کاربر فعلی سیستم (که لاگین کرده) بازیابی می‌شود.

توسط دستور new Claim(claimtype, claimvalue, ClaimValueTypes.String) یک شیء claim ساخته می‌شود که نشان می‌دهد مثلاً کاربر نقش خاصی دارد یا اطلاعات اضافی‌ای برای او ثبت شده و متد AddClaimAsync، claim ساخته‌شده را به لیست claimهای کاربر اضافه می‌کند .

سپس اگر عملیات موفق باشد (result.Succeeded)، کاربر به صفحه "Index" منتقل می‌شود.  
اگر مشکلی در افزودن claim باشد، همان صفحه‌ی فعلی دوباره نمایش داده می‌شود تا کاربر بتواند اصلاح کند.

View متد create به صورت زیر خواهد بود

@{

ViewData["Title"] = "Create new Claim";

}

<h2>@ViewData["Title"]</h2>

<**form** **asp-area**="Admin" **asp-controller**="Claim" **asp-action**="Create" method="post">

<div class="form-group">

<label for="claimtype"> Claim Type</label>

<input type="text" name="claimtype" class="form-control" required />

</div>

<div class="form-group">

<label for="claimvalue"> Claim Value</label>

<input type="text" name="claimvalue" class="form-control" required />

</div>

<button type="submit" class="btn btn-primary mt-3">Create</button>

</**form**>

**پارامترها:**

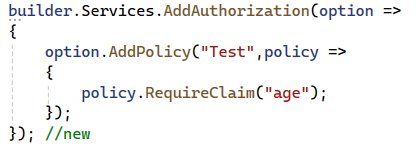
**Claimtype** نوع ادعا است، مثلاً "role" یا "email". مشخص می‌کند این claim درباره‌ی چه چیزی است.

**Claimvalue** مقدار ادعا است مثلاً اگر نوع "role" باشد، مقدار می‌تواند "admin" یا "student" باشد.

**ClaimValueTypes.String** نوع داده‌ی مقدار را مشخص می‌کند. در اینجا می‌گوییم مقدار از نوع رشته‌ای (string) است.

? میخواهیم یک Claim ایجاد کنیم با Claimtype به نام age و Claimvalue با مقدار15. این Claim در هنگام لاگین کردن با توجه به سن افراد عمل ... را انجام میدهد.

درمرحله اول یک Claim جدید ایجاد کنید با مشخصات بالا، سپس به program.cs رفته و یک policy برای Claim ایجاد میکنیم:

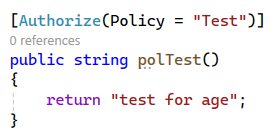


**1**

**2**

1. Test نام Policy است.
2. Claimtype را به عنوان ورودی دریافت می کند.

میتوانید اکشن متدی در کنترلر Home برای Policy ایجاد کنید:



**بخش دهم (API)**

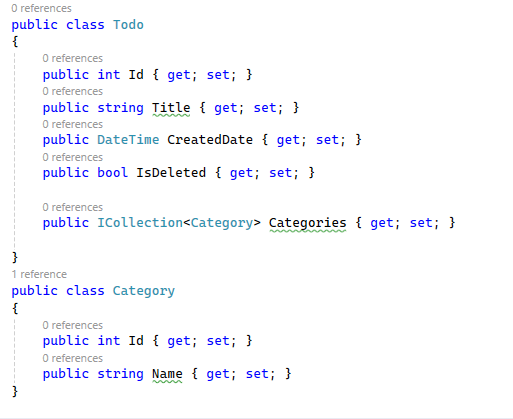
**API** (رابط برنامه‌نویسی کاربردی) مجموعه‌ای از قوانین و پروتکل‌هاست که به نرم‌افزارها اجازه می‌دهد با یکدیگر ارتباط برقرار کنند و از عملکردها یا داده‌های یکدیگر استفاده کنند**.**

برای ساخت یک API مراحل زیر را طی کنید:

* در منوی create new project در قسمت بندی ها Web API را انتخاب کنید و پروژه ای از نوع ASP .net core Web API را ایجاد کنید:

پروژه نیاز به پوشه view ندارد و ممکن است پوشه model نداشته باشد. که میتوانیم آن را ایجاد کنیم.

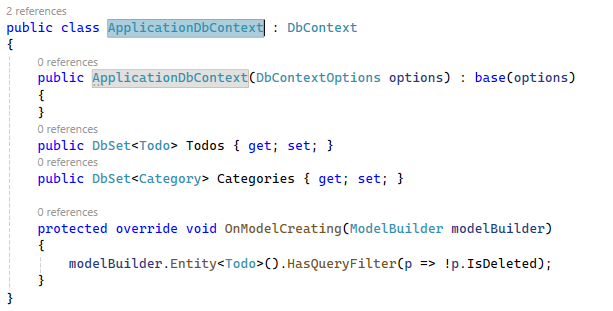
Entity به نام ToDo ایجاد کنید با خصوصیات زیر:



💡) اتصال دو جدول به صورت خودکار انجام میشود.

* در مرحله بعد باید ApplicationDbContext را ایجاد کنیم.

یکی از ویژگی های کلاس todo یک متغیر bool است به نام isDelete که موجب میشود تا کارهایی که توسط کاربر حذف شده اند از سیستم پاک نشوند بلکه در صفحه به نمایش در نیایند. به عبارتی فیلتر شوند. برای فیلتر کردن نیاز به دسترسی به پایگاه داده است:



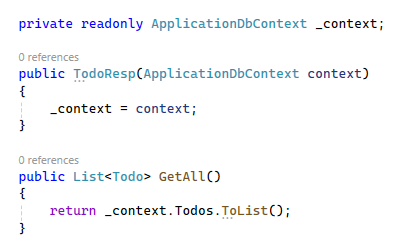
این کد در متد OnModelCreating یک فیلتر کوئری برای موجودیت Todo تعریف می‌کند که به طور پیش‌فرض فقط رکوردهای غیرحذف‌شده (یعنی رکوردهایی که IsDeleted آنها برابر با false است) را برمی‌گرداند.

کلمه کلیدی override به این معناست که متد OnModelCreating در حال بازنویسی (override) یک متد از کلاس پایه (پدر) است. این به برنامه‌نویس اجازه می‌دهد تا رفتار پیش‌فرض کلاس پایه را تغییر دهد یا آن را گسترش دهد.

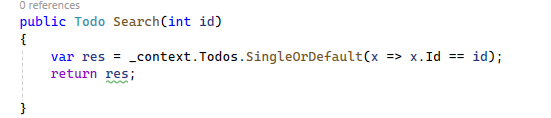
متد HasQueryFilter به منظور تعریف فیلترهای پیش‌فرض برای کوئری‌ها استفاده می‌شود. این متد به موجودیت‌ها این امکان را می‌دهد که شرایط خاصی را برای بازیابی داده‌ها در نظر بگیرند، بدون اینکه نیاز به اضافه کردن شرایط در هر کوئری باشد.

پوشه ای به نام service در model ایجاد میکنیم و کلاسی به نام TodoRepository میسازیم.

این کلاس عملیات لازم که در کنترلر نوشته میشد را پیاده سازی میکند و به خلوت شدن کنترلر کمک میکند:

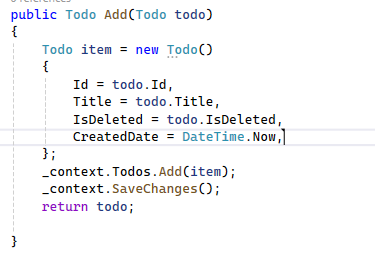
* ما به تابعی برای نمایش همه کارها نیاز داریم که نام آن را GetAll میگذاریم:

خروجی این تابع لیستی از کارهاست.

* تابع بعدی برای سرچ در بین کارها استفاده میشود:

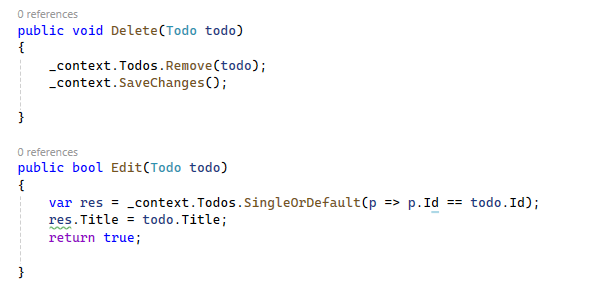
تابع سرچ یک id دریافت می کند و توسط تابع SingleOrDefault آن id را با id های موجود در جدول مقایسه میکند و درصورت وجود آن را برمیگرداند.

* تابع بعدی تابع اضافه کردن به کارهاست:



تابع Add یک Todo دریافت میکند و در خروجی هم یک Todo برمیگرداند. نمونه ای از کلاس Todo میسازیم و به ویژگی ها مقدار اولیه میدهیم. سپس نمونه را به جدول اضافه میکنیم.

* سپس برای حذف از جدول توابع زیر را مینویسیم:



**بخش یازدهم (آنلاین شاپ – پروژه محور)**

**قسمت اول: DataBase First**

عنوان "Code First" در چارچوب‌های ORM (Object-Relational Mapping) شناخته می‌شود. در این روش، ابتدا مدل‌های داده (کلاس‌ها) را تعریف می‌کنید و سپس با استفاده از کد، پایگاه داده را بر اساس این مدل‌ها ایجاد می‌کنید.

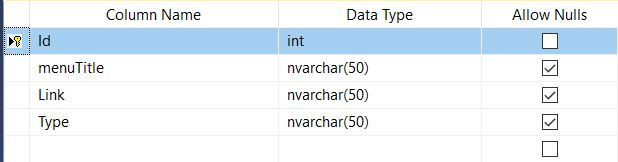
در ASP.NET، این فرآیند به شما این امکان را می‌دهد که بدون نیاز به نوشتن اسکریپت‌های SQL برای ایجاد جداول و روابط، به راحتی پایگاه داده خود را مدیریت کنید.

روش "DB First" یک رویکرد در توسعه نرم‌افزار است که در آن ابتدا پایگاه داده طراحی و ایجاد می‌شود و سپس از آن برای تولید مدل‌های داده (کلاس‌ها) استفاده می‌شود. این روش معمولاً در چارچوب‌های ORM (Object-Relational Mapping) مانند Entity Framework در ASP.NET به کار می‌رود.

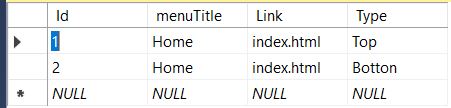
در مرحله اول یک پایگاه داده به نام OnlineShop ایجاد میکنیم. ابتدا شما یک پایگاه داده را با استفاده از ابزارهای مدیریت پایگاه داده (مانند SQL Server Management Studio) طراحی و ایجاد می‌کنید. جداول، روابط، اندیس‌ها و سایر عناصر پایگاه داده را تعریف می‌کنید.

💡) نوع داده ها و کلید اصلی را حتما به درستی مشخص و بررسی کنید.

می خواهیم برای منو یک مدل در پایگاه داده بسازیم.

کلاس منو دارای ویژگی ID ، عنوان ، لینک و نوع است. نوع منو میتواند Top یا Botton باشد.

میتوانید مقدار اولیه نیز به فیلد ها بدهید.



در مرحله بعد باید جدول را به پروژه اضافه کنیم. برای انجام این کار بااید دستور scaffold را در console به جای migration وارد کنیم.

Scaffold-DbContext “server=DESKTOP-T89EF7E;Database=Entesharat;Trusted\_Connection=True;TrustServerCertificate=true” Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models/DB -Tables Menus

پس از وارد کردن این دستور مدل ایجاد میشود . همچنین یک DbContext هم به صورت خودکار ایجاد میشود. که باید آن را در program.cs ، build کنیم:

builder.Services.AddDbContext<EntesharatContext>();

برای اضافه کردن جداول به پروژه میتوان مجددا از scaffold استفاده کرد.

**قسمت دوم: Area**

در مرحله بعد به یک area برای جدا سازی بخش user و admin داریم. برای ساخت Area مسیر زیر را بروید:

Add> new scaffold > new area > name

💡) همه پکیج ها را به اخرین نسخه آپدیت کنید تا به خطا بر نخورید.

سپس یک map controller routeدر program.cs ایجاد کنید:

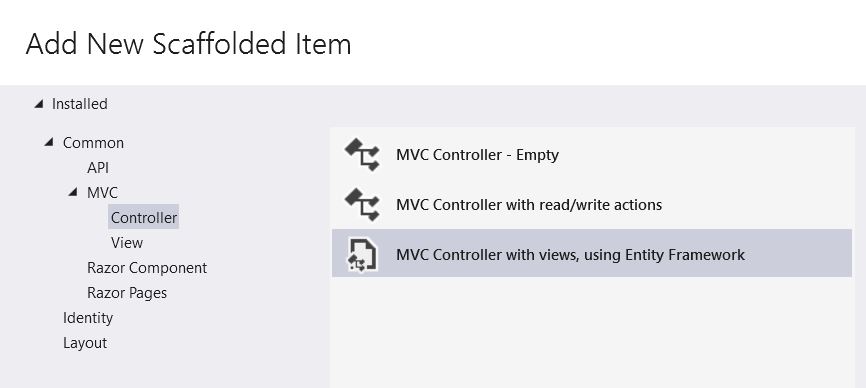
app.MapControllerRoute(

name: "areas",

pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

💡) چون فایل‌های ViewStart.cshtml و ViewImports.cshtml نقش مهمی در ساختار Razor دارن، مخصوصاً وقتی Area استفاده می‌کنیم. محتویات پوشه Shared و دو فایل گفته شده را به area نیز کپی کنید.

سپس کنترلری برای منو میسازیم. برای اینکه کنترلر پیش ساخته باشد در زمان ساخت مسیر زیر را بروید:



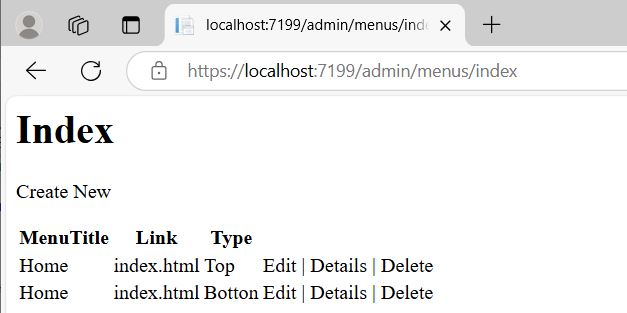
این گزینه یک کنترلر با توجه به مدل به همراه View و پوشه های دیگر میسازد.

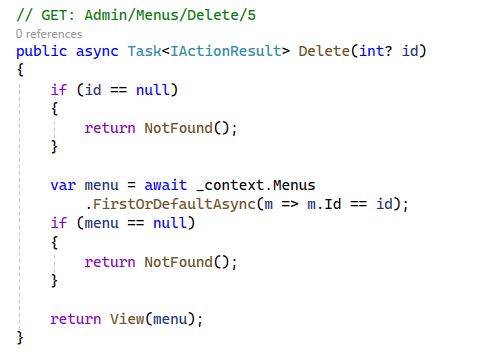
💡 تکمیلی) کنترلرهایی که داخل Area هستند مثل UserController در Areas/Admin/Controllers/ باید این اتریبیوت بالا داشته باشند:

[Area("Admin")]

public class UserController : Controller{…}

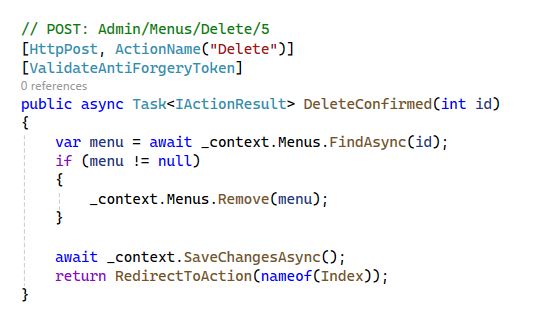
کنترلر عملیات CRUD را پیاده سازی کرده و اگر آدرس زیر را سرچ کنید صفحه index منو را خواهید دید.



برای حذف آیتم های منو دو تابع در کنترلر نوشته شده. تابع اول به دنبال id آیتم در جدول منو میگردد و تابع دوم به حذف آیتم پیدا شده میپردازد:

این تابع id را دریافت کرده و در صورت وجود به دنبال ظان در جدول منو میگردد. از کلمه کلیدی async استفاده شده تا تابع با مکث اجرا شود.

تابع بعدی بررسی میکند آیا مایل به حذف آیتم هستید؟ این تابع و تابع قبلی در اصل یک تابع اند و نام های متفاوتی دارند.

Action name را Delete که نام تابع قبلی بود میزاریم تا هردو تابع یک عمل را انجام دهند و id پیدا شده در تابه اول به عنوان وردی به تابع دوم داده شود.

همچنین تابع را از نوع Async قرار دادیم تا زمانی که چندین درخواست همزمان ارسال شد دچار خطا نشویم. همچنین از save change Async نیز باید استفاده کنیم.

* حالا با وارد کردن نام تابع ها پس از admin/menus/ میتوانیم به آن ها دست پیدا کنیم.

**قسمت سوم: inject**

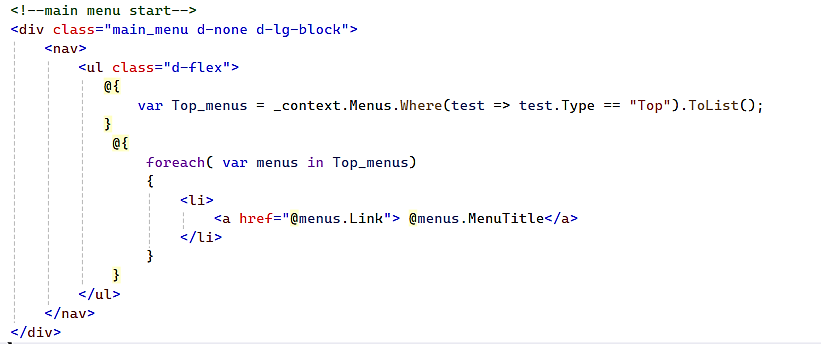
توسط دستور inject میتوانیم داده ها یا سرویس خاصی را به view تزریق کنیم.

میخواهیم در منو اصلی عنوان های منو نمایش داده شود و سپس با کلیک بر روی آن به لینک مورد نظر وارد شویم. برای انجام این کار در مرحله اول به جدول منو نیاز داریم.

💡) منو باید در همه جای وبسایت قابل مشاهده باشد پس دستورات مربوط به view آن را در فایل \_layout می نویسیم.

دستور زیر را برای ساخت نمونه ای از جدول در ابتدای دستور های دیگر وارد کنید:

@inject OnlineShop.Models.DB.OnlineShopContext \_context;

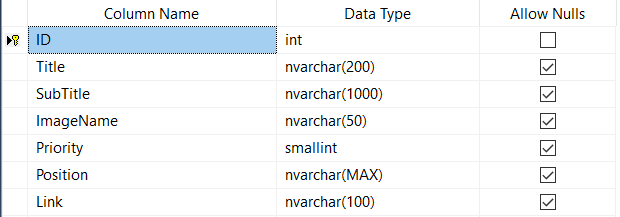
در مرحله بعد در منو اصلی (main menu) عنوان های منویی که از نوع top را نمایش دهیم. به دستورات html مربوط به main menu رفته و دستورات زیر را اضافه کنید:

پس از پیدا کردن منو های از نوع top لیستی از لینک ها و عنوان آن ها میسازیم.

**قسمت چهارم: Slider**

قالب سایت یک اسلایدر در صفحه اصلی و سپس دو بنر کنار هم دارد. می خواهیم ادمین بتواند توسط صفحه ای عکس هارا ویرایش یا حذف یا اضافه کند. برای انجام این کار نیاز به جدولی به نام banner داریم که شامل نام عکس ، اولویت عکس و محل قرارگیری عکس (اسلایدر یا بنر اول یا بنر دوم) است.

جدول را در پایگاه داده ایجاد کرده و در پروژه توسط scaffold مدل ها را ایجاد می کنیم.



💡) نوع داده smallint برای نگه داری اعداد کوچک استفاده می شود.

سپس توسط دستور scaffold مدل را ایجاد میکنیم.

Scaffold-DbContext "server=DESKTOPT89EF7E;Database=OnlineShop;Trusted\_Connection=True;TrustServerCertificate=true" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models/DB -Force

💡) -Force جدیدترین جدول و تغییرات را اعمال میکند.

مدل بنر به یک کنترلر نیاز دارد تا عمل crud روی بنر انجام شود. کنترلری پیش ساخته برای بنر درست کنید. (در ناحیه ادمین)

می خواهیم ادمین در هنگام کار با عکس پیش نمایشی از عکس داشته باشد و در صفحه مدیریت بنر بتواند از فایل های خودش عکسی را بارگذاری کند.

برای اینکه در هنگام ساخت بنر توسط ادمین یا ادیت آن ، ادمین بتواند یک فایل دریافت کند باید در صفحه view اکشن create در کنترلر Banner دستور زیر را در بخش مربوط به دریافت عکس وارد کنید.

<input type="file" asp-for="ImageName" class="form-control" /> ⨯

<input type="file" name="ImageFile" class="form-control" /> ✓

ویژگی ImageName در مدل از جنس String است و نام عکس را دریافت می کند. اما تگ input از ورودی یک فایل که عکس است را میگیرد. پس این تگ نیاز به asp-for ندارد و به جای آن از name="ImageFile" استفاده می کنیم.

سپس باید position را به صورت کمبو باکس به ادمین نشان دهیم تا هنگام اضافه کردن بنر مشخص کند بنر برای اسلایدر است یا بنر اول یا بنر دوم. به جای input از select استفاده می کنیم:

<div class="form-group">

<label asp-for="Position" class="control-label"></label>

<select asp-for="Position">

<option value="Slider">Slider</option>

<option value="Banner1">Banner1</option>

<option value="Banner2">Banner2</option>

</select>

<span asp-validation-for="Position" class="text-danger"></span>

</div>

در مرحله قبل توسط input عکس بنر را دریافت کردیم. این عکس باید در پوشه img در wwwroot قرار بگیرد. عکس توسط فرم و تگ input دریافت میشود و سپس برای ذخیره سازی در پروژه و پایگاه داده باید به کنترلر ارسال شود. فرمی که فایلی دریافت میکند باید نوعش مشخص شود. همچنین نام Area که کنترلر در آن وجود دارد را نیز به form اضافه می کنیم:

<**form** **asp-action**="Create" **asp-controller**="Banners" **asp-area**="Admin" enctype="multipart/form-data" method="post">

دستورات مربوط به دریافت مقادیر ویژگی های بنر در فرم قرار میگیرد.

توسط فرم اطلاعات را دریافت کرده و آن ها را به اکشن متد create میفرستیم. عکس را دریافت و آن را در پوشه wwwroot قرار میدهیم. سپس نام عکس را گرفته و یک کد Guid که بی همتا است به آن اضافه میکنیم.

به این صورت نام هر عکس با عکس جدید دیگر متفاوت خواهد بود. سپس نام را در imageName قرار میدهیم.



**4**

**2**

**1**

**3**

1. متد Bind به طور معمول برای ارتباط بین کنترل‌های وب و مدل‌های داده استفاده می‌شود. این متد به شما اجازه می‌دهد تا داده‌ها را به سادگی از یک منبع داده (مثل پایگاه‌داده) به کنترل‌های وب مانند GridView، FormView و ... متصل کنید. سپس اگر فایل عکس دریافت شده بود این خط کد یک نام جدید برای تصویر ایجاد می‌کند. ابتدا یک GUID جدید تولید می‌شود (که یک شناسه یکتا است) و سپس پسوند فایل تصویر (مثل .jpg یا .png) از نام فایل اصلی استخراج می‌شود. در نهایت، این دو به هم متصل می‌شوند تا یک نام یکتای جدید برای تصویر بسازند.
2. این کد مسیر کامل یک تصویر را در دایرکتوری جاری برنامه ایجاد می‌کند. با استفاده از نام تصویر (banner.ImageName)، مسیر نهایی به پوشه wwwroot img slider ترکیب می‌شود. مسیر پروژه + مسیر عکس+ نام فایل
3. این کد یک فایل تصویر را از ورودی (ImageFile) به یک مسیر مشخص (ImagePath) کپی می‌کند. اگر فایل در آن مسیر وجود نداشته باشد، آن را ایجاد می‌کند.
4. برای آدرس دهی به کنترلری در یک Area ، باید نام آن را ذکر کنیم.

در مرحله بعد میخواهیم در پنل مدیریت بنر عکس ها نمایش یابند. در متد index کنترلر بنر دستورات زیر را وارد می کنیم. و توسط یک foreach عکس هارا به نمایش در می آوریم.

@model IEnumerable<OnlineShop.Models.DB.Banner>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<**a** **asp-action**="Create">Create New</**a**>

</p>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Title)

</th>

<th>

@Html.DisplayNameFor(model => model.SubTitle)

</th>

<th>

picture

</th>

<th>

@Html.DisplayNameFor(model => model.Priority)

</th>

<th>

@Html.DisplayNameFor(model => model.Position)

</th>

<th>

@Html.DisplayNameFor(model => model.Link)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.SubTitle)

</td>

<td>

<img class="img-thumbnail" style="max-width:100px" src="/img/slider/@item.ImageName" />

</td>

<td>

@Html.DisplayFor(modelItem => item.Priority)

</td>

<td>

@Html.DisplayFor(modelItem => item.Position)

</td>

<td>

@Html.DisplayFor(modelItem => item.Link)

</td>

<td>

<**a** **asp-action**="Edit" **asp-route-id**="@item.Id">Edit</**a**>

<**a** **asp-action**="Details" **asp-route-id**="@item.Id">Details</**a**>

<**a** **asp-action**="Delete" **asp-route-id**="@item.Id">Delete</**a**>

</td>

</tr>}

</tbody>

</table>

میتوانیم همین کار را برای Edit هم انجام دهیم. وقتی ادمین میخواهد بنر را ویرایش کند عکس فعلی بنر را مشاهده کند. و با دکمه ای عکس جدیدی انتخاب کند.

پس به ویو Edit در کنترلر بنر رفته و در دستوری که برای عکس نوشته شده تغیرات زیر را اضافه کنید.

<input type="file" asp-for="ImageName" class="form-control" />

<img class="img-thumbnail" style="max-width:100px" src="/img/slider/@Model.ImageName" />

Input جهت دریافت عکس جدید و img جهت دیدن عکس فعلی استفاده می شود.

💡) برای دریافت عکس نوع فرم باید enctype="multipart/form-data" باشد.

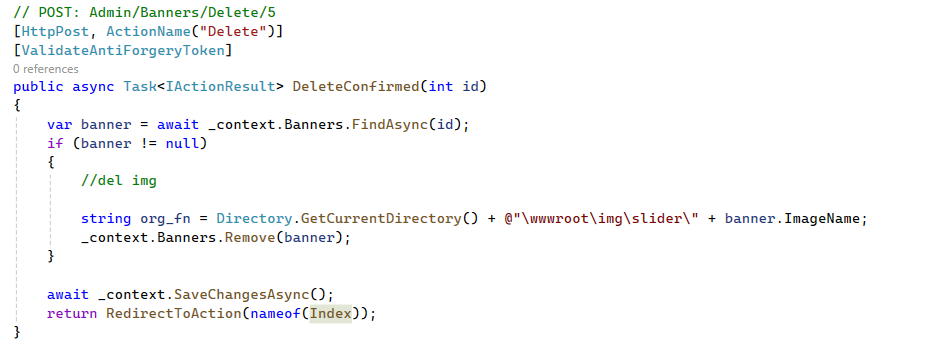
همچنین می خواهیم اگر ادمین وارد صفحه شد اما تغییراتی در عکس ایجاد نکرد، همان عکس قبلی نمایش یابد. به همین دلیل از در hidden در ابتدای دستورات استفاده می کنیم تا مقدار id و نام عکس را به صورت مخفی دریافت کنیم.

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

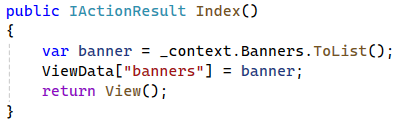
<input type="hidden" asp-for="ImageName" />

حالا اگر بخواهیم توسط تابع Del بنری را پاک کنیم ، دستوررات فقط نام عکس را از پایگاه داده پاک می کند اما ما میخواهیم خود عکس هم از wwwroot حذف شود.

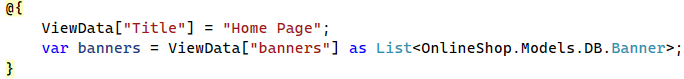
در مرحله اول در صفحه حذف عکس را توسط دستورات قبلی نمایش می دهیم تا ادمین ابتدا عکس را ببیند.

در کنترلر دستوری در تابع حذف می نویسیم برای کار با فایل ها و حذف آن ها. برای انجام این کار از تابعی که مسیر پروژه + مسیر عکس را بخواهد استفاده می کنیم:

برای نمایش عکس اسلایدر و بنر در صفحه اصلی به متد index درکنترلر home رفته و لیستی از بنر هارا درون متغیری ریخته و توسط ViewData آن هارا به view منتقل می کنیم:



سپس در ویو آن را دریافت می کنیم:



در ویو index به دنبال بخش مربوط به اسلایدر میگردیم و با یک شرط چک می کنیم که لیست بنر ها خالی نباشد. در این صورت متغیری میسازیم و در لیست بنر ها بنرهایی با پوزیشن اسلایدر را مجددا لیست کرده آن هارا بر اساس اولویت مرتب می کنیم و سپس در متغیر میریزیم.

@if (banners != null)

{

var slider = banners.Where(b => b.Position == "Slider").OrderBy(o => o.Priority).ToList();

foreach (var item in slider)

{

<div class="single\_slider d-flex align-items-center" data-

bgimg="assets/img/slider/@item.ImageName">

<div class="container">

<div class="row">

<div class="col-lg-6 col-md-7">

<div class="slider\_text">

<span>Lookbook</span>

<h1>@item.Title</h1>

<p>@item.SubTitle</p>

<a class="btn btn-primary" href=@item.Link>Explore Now</a>

</div>

</div>

</div>

</div>

</div>

}

}